Mastering Object-Oriented Python

Second Edition

Build powerful applications with reusable code using OOP design patterns and Python 3.7



Steven F. Lott

Packt
www.packt.com

Mastering Object-Oriented Python
Second Edition

Build powerful applications with reusable code using OOP design patterns and Python 3.7

Steven F. Lott

BIRMINGHAM - MUMBAI

Mastering Object-Oriented Python Second Edition

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Richa Tripathi
Acquisition Editor: Chaitanya Nair
Content Development Editor: Zeeyan Pinheiro
Senior Editor: Afshaan Khan
Technical Editor: Ketan Kamble
Copy Editor: Safis Editing
Project Coordinator: Vaidehi Sawant
Proofreader: Safis Editing
Indexer: Manju Arasan
Production Designer: Jayalaxmi Raja

First published: April 2014
Second edition: June 2019

Production reference: 1130619

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78953-136-7

www.packtpub.com

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.packt.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `customercare@packtpub.com` for more details.

At `www.packt.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Steven F. Lott has been programming since the 1970s, when computers were large, expensive, and rare. As a contract software developer and architect, he has worked on hundreds of projects, from very small to very large ones. He's been using Python to solve business problems for over 10 years. His other titles with Packt include *Python Essentials*, *Mastering Object-Oriented Python*, *Functional Python Programming Second Edition*, *Python for Secret Agents*, and *Python for Secret Agents II*. Steven is currently a technomad who lives in various places on the East Coast of the US. You can follow him on Twitter via the handle `@s_lott`.

About the reviewers

Cody Jackson is a disabled military veteran, the founder of Socius Consulting – an IT and business management consulting company in San Antonio – and a cofounder of Top Men Technologies. He is currently employed at CACI International as the lead ICS/SCADA modeling and simulations engineer. He has been involved in the tech industry since 1994, when he joined the Navy as a nuclear chemist and Radcon Technician. Prior to CACI, he worked at ECPI University as a computer information systems adjunct professor. A self-taught Python programmer, he is the author of *Learning to Program Using Python* and *Secret Recipes of the Python Ninja*. He holds an Associate in Science degree, a Bachelor of Science degree, and a Master of Science degree.

Hugo Solis is an assistant professor in the Physics Department at the University of Costa Rica. His current research interests are computational cosmology, complexity, cryptography, and the influence of hydrogen on material properties. He has vast experience with languages, including C/C++ and Python for scientific programming. He is a member of the Free Software Foundation and has contributed code to some free software projects. He has also been a technical reviewer for *Hands-On Qt for Python Developers* and *Learning Object-Oriented Programming*, and he is the author of the *Kivy Cookbook* from Packt Publishing. Currently, he is in charge of the IFT, a Costa Rican scientific nonprofit organization for the multidisciplinary practice of physics.

I'd like to thank Katty Sanchez, my beloved mother, for her support and vanguard thoughts.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

This book will introduce you to many advanced features of the Python programming language. The focus is on creating the highest quality Python programs possible. This requires exploring design alternatives and determining which design offers the best performance while still being a good fit for the problem that is being solved.

The majority of this book showcases a number of alternatives for a given design. Some will offer better performance, while some will appear simpler or be a better solution to the problem domain. It's essential to locate the best algorithms alongside optimal data structures in order to create the most value with the least computer processing. Time is money, and programs that save time will create more value for their users. Python makes a number of internal features directly available to our application programs. This means that our programs can be very tightly integrated with existing Python features. We can leverage numerous Python features by ensuring that our **object-oriented designs** (**OODs**) integrate well.

As we explore different algorithms and data structures, we'll discover different memory and performance alternatives. It's an important OOD skill to be able to work through alternate solutions in order to properly optimize the final application. One of the more important themes of this book is that there's no single best approach to any problem.

As many of the examples as possible have full type hints. A few of the examples rely on packages outside the standard library, where you'll find that type hints are either missing or are incomplete. The examples have to be processed with the **mypy** tool to confirm the types are used consistently.

As we move toward achieving mastery of object-oriented Python, we'll spend a great deal of time reading Python code from a variety of sources. We'll observe wide variability even within the Python standard library modules. Rather than presenting examples that are all perfectly consistent, we've opted for some inconsistency; the lack of consistency will help to read kinds of code, as seen in

various open source projects encountered in the wild.

Who this book is for

This book uses advanced Python. You'll need to be relatively familiar with Python 3. It helps to learn a programming language when you have a problem of your own to solve.

If you are a skilled programmer in other languages, then you may find this book useful if you want to switch to Python. Note that this book doesn't introduce any syntax or other foundational concepts.

Python 2 programmers may find this particularly helpful when they switch to Python 3. We won't cover any of the conversion utilities (such as the **2to3** tool) or any of the coexistence libraries (such as the `six` module). This book is focused on new developments entirely in Python 3.

What this book covers

In this book, we'll cover three broad areas of advanced Python topics. Each topic will be broken into a series of chapters examining a variety of details.

`Section 1`, *Tighter Integration via Special Methods*, looks at **object-oriented programming** (**OOP**) techniques in depth and how we can more tightly integrate the class definitions of our applications with Python's built-in features. This section consists of nine chapters, which are as follows:

- `Chapter 1`, *Preliminaries, Tools, and Techniques*, covers some preliminary topics, such as `unittest`, `doctest`, `docstring`, and some special method names.
- `Chapter 2`, *The _init_() Method*, provides us with a detailed description and implementation of the `_init_()` method. We will examine different forms of initialization for simple objects. Following this, we can explore more complex objects that involve collections and containers.
- `Chapter 3`, *Integrating Seamlessly – Basic Special Methods*, explains, in detail, how we can expand a simple class definition to add special methods. We'll need to take a look at the default behavior inherited from the object so that we can understand what overrides are required and when they're actually required.
- `Chapter 4`, *Attribute Access, Properties, and Descriptors*, explores how default processing works in some detail. Here, we will learn how to decide where and when to override the default behavior. We will also explore descriptors and gain a much deeper understanding of how Python's internals work.
- `Chapter 5`, *The ABCs of Consistent Design*, examines the abstract base classes in the `collections.abc` module. In this chapter, we'll look at the general concepts behind the various containers and collections that we might want to revise or extend. Similarly, we'll look at the concepts behind the numbers that we might want to implement.
- `Chapter 6`, *Using Callables and Contexts*, uncovers several ways to create context managers using the tools in `contextlib`. We'll demonstrate a number of variant designs for callable objects. This will show you why a stateful callable object is sometimes more useful than a simple function. We'll also explore how to use some of the existing Python context managers before we

dive in and write our own context manager.

- `Chapter 7`, *Creating Containers and Collections*, focuses on the basics of container classes. We'll review the variety of special methods that are involved in creating a container and the various features that containers offer. We'll address extending built-in containers to add features. We'll also look at wrapping built-in containers and delegating methods through the wrapper to the underlying container.
- `Chapter 8`, *Creating Numbers*, covers these essential arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, and `**`. We'll also explore these comparison operators: `<`, `>`, `<=`, `>=`, `==`, and `!=`. We'll finish by summarizing some of the design considerations that go into extending or creating new numbers.
- `Chapter 9`, *Decorators and Mixins – Cross-Cutting Aspects*, covers simple function decorators, function decorators with arguments, class decorators, and method decorators.

`Section 2`, *Object Serialization and Persistence*, explores a persistent object that has been serialized to a storage medium; perhaps it's transformed to JSON and written to the filesystem. An ORM layer can store the object in a database. This section examines the alternatives for handling persistence. It contains five chapters, which are as follows:

- `Chapter 10`, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML*, covers simple persistence using libraries focused on various data representations such as JSON, YAML, pickle, XML, and CSV.
- `Chapter 11`, *Storing and Retrieving Objects via Shelve*, explains basic database operations with Python modules, such as shelve (and dbm).
- `Chapter 12`, *Storing and Retrieving Objects via SQLite*, uncovers the more complex world of SQL and the relational database. Because SQL features don't match OOP features well, we have an impedance mismatch problem. A common solution is to use ORM to allow us to persist a large domain of objects. The SQLAlchemy package will be used as an example of the many ORMs that are available.
- `Chapter 13`, *Transmitting and Sharing Objects*, looks at the HTTP protocol, JSON, YAML, and XML representations to transmit an object.
- `Chapter 14`, *Configuration Files and Persistence*, covers various ways in which a Python application can work with a configuration file.
- `Chapter 15`, *Design Principles and Patterns*, reviews the **SOLID** design principles. These can help organize high-quality, maintainable Python software by following some best practices.

Section 3, *Object-Oriented Testing and Debugging*, shows you how to gather data to support and debug your own high-performance programs. It includes information on creating the best possible documentation in order to reduce the confusion and complexity of the support. This section contains the final five chapters, which are as follows:

- Chapter 16, *The Logging and Warning Modules*, looks at using the logging and warning modules to create audit information, as well as debugging. Additionally, we'll take a significant step beyond using the `print()` function.
- Chapter 17, *Designing for Testability*, covers designing for testability and demonstrates how to use `unittest` and `doctest`.
- Chapter 18, *Coping with the Command Line*, looks at using the `argparse` module to parse options and arguments. We'll take this a step further and use the command design pattern to create program components that can be combined and expanded without resorting to writing shell scripts.
- Chapter 19, *Module and Package Design*, covers module and package design. This is a higher-level set of considerations; we'll take a look at related classes in a module and related modules in a package.
- Chapter 20, *Quality and Documentation*, explores how we can document our design to create some kind of trust that our software is correct and has been properly implemented.

To get the most out of this book

In order to compile and run the examples included in this book, you will require the following software:

- Python Version 3.7 or higher, with the standard suite of libraries:
  - We'll use **mypy** to check type hints (`http://mypy-lang.org`).
- We'll take a look at these additional packages:
  - PyYAML (`http://pyyaml.org`).
  - SQLAlchemy (`http://www.sqlalchemy.org`): When building this, check the installation guide carefully. In particular, refer to `https://docs.sqlalchemy.org/en/12/intro.html#installing-the-c-extensions` for information on simplifying the installation by disabling the C extension.
  - Flask (`http://flask.pocoo.org`).
  - Requests (`https://2.python-requests.org/en/master/`).
  - Jinja (`http://jinja.pocoo.org/`).
  - PyTest (`https://docs.pytest.org/en/latest/`).
  - Sphinx (`http://sphinx-doc.org`).
- Optionally, you might want to use the **Black** tool to format your code consistently (`https://black.readthedocs.io/en/stable/`).
- Additionally, the overall test suite for this book's code is run using the **tox** tool (`https://tox.readthedocs.io/en/latest/`).

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of the following:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Object-Oriented-Python-Second-Edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Code in Action

Visit the following link to see the code being executed:

http://bit.ly/2XIu8Tk

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
def F(n: int) -> int:
    if n in (0, 1):
        return 1
    else:
        return F(n-1) + F(n-2)
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
def factorial(n: int) -> int:
    """Compute n! recursively.

    :param n: an integer >= 0
    :returns: n!

    Because of Python's stack limitation, this won't
    compute a value larger than about 1000!.

    >>> factorial(5)
    120
    """
```

Any command-line input or output is written as follows:

```
$ python3 -m pip install --upgrade pip
$ python3 -m pip install black
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select System info from the Administration panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

Section 1: Tighter Integration Via Special Methods

We'll extend the core object-oriented programming techniques to allow for increased integration of the classes we create with other features of Python.

The following chapters will be covered in this section:

- `Chapter 1`, *Preliminaries, Tools, and Techniques*
- `Chapter 2`, *The __init__() Method*
- `Chapter 3`, *Integrating Seamlessly – Basic Special Methods*
- `Chapter 4`, *Attribute Access, Properties, and Descriptors*
- `Chapter 5`, *The ABCs of Consistent Design*
- `Chapter 6`, *Using Callables and Contexts*
- `Chapter 7`, *Creating Containers and Collections*
- `Chapter 8`, *Creating Numbers*
- `Chapter 9`, *Decorators and Mixins – Cross-Cutting Aspects*

Preliminaries, Tools, and Techniques

To make the design issues in the balance of the book more clear, we need to look at some the problems that serve as motivation. One of these is using **object-oriented programming** (**OOP**) for simulation. Simulation was one of the early problem domains for OOP. This is an area where OOP works out particularly elegantly.

We've chosen a problem domain that's relatively simple: the strategies for playing the game of *blackjack*. We don't want to endorse gambling; indeed, a bit of study will show that the game is stacked **heavily** against the player. This should reveal that most casino gambling is little more than a tax on the innumerate.

The first section of this chapter will review the rules of the game of *Blackjack*. After looking at the card game, the bulk of this chapter will provide some background in tools that are essential for writing complete Python programs and packages. We'll look at the following concepts:

- The Python runtime environment and how the special method names implement the language features
- **Integrated Development Environments** (**IDEs**)
- Using the `pylint` or `black` tools to create a uniform style
- Using type hints and the `mypy` tool to establish proper use of functions, classes, and variables
- Using `timeit` for performance testing
- Using `unittest`, `doctest`, and `pytest` for unit testing
- Using `sphinx` and RST-based markup to create usable documentation

While some of these tools are part of the Python standard library, most of them are outside the library. We'll discuss installation of tools when we talk about the Python runtime in general.

This book will try to avoid digressing into the foundations of Python OOP. We're assuming that you've already read Packt's *Python3 Object-Oriented Programming*. We don't want to repeat things that are nicely stated elsewhere.

We will focus on Python 3.

We'll refer to a number of common object-oriented design patterns and will try to avoid repeating the presentation in Packt's *Learning Python Design Patterns*.

We'll cover the following topics in this chapter:

- About the Blackjack game
- The Python runtime and special methods
- Interaction, scripting and tools
- Selecting an IDE
- Consistency and style
- Type hints and the `mypy` program
- Performance – the `timeit` module
- Testing – `unittest` and `doctest`
- Documentation – `sphinx` and RST markup
- Installing components

Technical requirements

The code files for this chapter can be found at `https://git.io/fj2UB`.

About the Blackjack game

Many of the examples in the book will center on simulations of a process with a number of moderately complex state changes. The card game of *Blackjack* involves a few rules and a few state changes during play. If you're unfamiliar with the game of *Blackjack*, here's an overview.

The objective of the game is to accept cards from the dealer to create a hand that has a point total that is between the dealer's total and twenty-one. The dealer's hand is only partially revealed, forcing the player to make a decision without knowing the dealer's total or the subsequent cards from the deck.

The number cards (2 to 10) have point values equal to the number. The face cards (Jack, Queen, and King) are worth 10 points. The Ace is worth either eleven points or one point. When using an ace as eleven points, the value of the hand is *soft*. When using an ace as one point, the value is *hard*.

A hand with an Ace and a seven, therefore, has a hard total of eight and a soft total of 18. This leads the player to choose to take extra cards. If the dealer is showing a face card, it's very likely the dealer is holding twenty points, and the player may not want to risk taking another card.

Each suit has four two-card combinations that total 21. These are all called *Blackjack,* even though only one of the four combinations involves a Jack. These combinations often provide a bonus payout, because there are only four of them available.

Most of the game is about proper choice of cards. There is, of course, a betting element. The distinction between playing and betting is made somewhat more complicated by the provision to split one hand into two hands. This is allowed when the player's two cards have the same rank. This option will be detailed in the next section on how the game is played.

Playing the game

The mechanics of play generally work as follows. The details can vary, but the outline is similar:

- First, the player and dealer each get two cards. The player, of course, knows the value of both of their cards. They're dealt face up in a casino.
- One of the dealer's cards is revealed to the player. It's displayed face up. The player, therefore, knows a little bit about the dealer's hand, but not everything. This is typical of more complex simulations where partial information is available and statistical modeling is required to make appropriate decisions.
- If the dealer has an Ace showing, the player is offered the opportunity to place an additional insurance bet. This is a special case, and is typical of more complex simulations where there are exceptions.
- For the balance of the game, the player can elect to receive cards, or stop receiving cards. There are four choices available:
  - The player can *hit*, which means take another card.
  - They player can or *stand* or *stand pat* with the cards dealt.
  - If the player's cards match, the hand can be split. This entails an additional bet, and the two hands are played separately.
  - The player can double their bet before taking one last card. This is called *doubling down*.

The final evaluation of the hand works as follows:

- If the player went over 21, the hand is a bust, the player loses, and the dealer's face-down card is irrelevant. This provides an advantage to the dealer.
- If the player's total is 21 or under, then the dealer takes cards according to a simple, fixed rule. The dealer must hit a hand that totals less than 18; the dealer must stand on a hand that totals 18 or more.
- If the dealer goes bust, the player wins.
- If both the dealer and player are 21 or under, the hands are compared. The higher total is the winner. In the event of a tie, the game is a *push*, neither a win nor a loss. If the player wins with 21, they win a larger payout, usually

1.5 times the bet.

The rules can vary quite a bit. We'll elide these details to focus on the Python code required for simulation.

Blackjack player strategies

In the case of *blackjack*, there are actually two kinds of strategies that the player must use:

- A strategy for deciding what play to make: take insurance, hit, stand, split, or double down.
- A strategy for deciding what amount to bet. A common statistical fallacy leads players to raise and lower their bets in an attempt to preserve their winnings and minimize their losses. These are interesting, stateful algorithms in spite of the underlying fallacies.

These two sets of strategies are, of course, prime examples of the **Strategy** design pattern.

Object design for simulating Blackjack

We'll use elements of the game, such as the player, hand, and card, as examples for object modeling. We won't design the entire simulation. We'll focus on elements of this game because they have some nuance, but aren't terribly complex.

The cards are relatively simple, immutable objects. There are a variety of modeling techniques available. Cards fall into a simple class hierarchy of the number cards, face cards, and the Ace. There are simple containers, including hands of card instances, and decks of cards as well. These are stateful collections with cards being added and removed. There are a number of ways to implement this in Python and we'll look at many alternatives. We also need to look at the player as a whole. A player will have a sequence of hands, as well as a betting strategy and a *Blackjack* play strategy. This is a rather complex composite object.

The Python runtime and special methods

One of the essential concepts for mastering object-oriented Python is to understand how object methods are implemented. Let's look at a relatively simple Python interaction:

```
>>> f = [1, 1, 2, 3]
>>> f += [f[-1] + f[-2]]
>>> f
[1, 1, 2, 3, 5]
```

We've created a list, `f`, with a sequence of values. We then mutated this list using the `+=` operator to append a new value. The `f[-1] + f[-2]` expression computes the new value to be appended.

The value of `f[-1]` is implemented using the list object's `__getitem__()` method. This is a core pattern of Python: the simple operator-like syntax is implemented by special methods. The special methods have names surrounded with __ to make them distinctive. For simple prefix and suffix syntax, the object is obvious; `f[-1]` is implemented as `f.__getitem__(-1)`.

The additional operation is similarly implemented by the `__add__()` special method. In the case of a binary operator, Python will try both operands to see which one offers the special method. In this example, both operands are integers, and both will provide a suitable implementation. In the case of mixed types, the implementation of the binary operator may coerce one value into another type. `f[-1] + f[-2]`, then, is implemented as `f.__getitem__(-1).__add__(f.__getitem__(-2))`.

The update of `f` by the `+=` operator is implemented by the `__iadd__()` special method. Consequently, `f += [x]` is implemented as `f.__iadd__([x])`.

Throughout the first eight chapters, we'll look very closely at these special methods and how we can design our classes to integrate very tightly with Python's built-in language features. Mastering the special methods is the essence of mastering object-oriented Python.

Interaction, scripting, and tools

Python is often described as *Batteries Included* programming. Everything required is available directly as part of a single download. This provides the runtime, the standard library, and the IDLE editor as a simple development environment.

It's very easy to download and install Python 3.7 and start running it interactively on the desktop. The example in the previous section included the `>>>` prompt from interactive Python.

If you're using the Iron Python (**IPython**) implementation, the interaction will look like this:

```
In [1]: f = [1, 1, 2, 3]
In [3]: f += [f[-1] + f[-2]]
In [4]: f
Out[4]: [1, 1, 2, 3, 5]
```

The prompt is slightly different, but the language is the same. Each statement is evaluated as it is presented to Python.

This is handy for some experimentation. Our goal is to build tools, frameworks, and applications. While many of the examples will be shown in an interactive style, most of the actual programming will be via script files.

Running examples interactively makes a profound statement. Well-written Python code should be simple enough that it can be run from the command line.

TIP ***Good Python is simple.*** *We should be able to demonstrate a design at the* `>>>` *prompt.*

Interactive use is not our goal. Exercising code from the `>>>` prompt is a quality test for complexity. If the code is too complex to exercise it from the `>>>` prompt, then refactoring is needed.

The focus of this book is on creating complete scripts, modules, packages, and applications. Even though some examples are shown in interactive mode, the

objective is to create Python files. These files may be as simple as a script or as complex as a directory with files to create a web application.

Tools such as `mypy`, `pytest`, and `pylint` work with Python files. Preparing script files can be done with almost any text editor. It's best, however, to work with an IDE, where a number of tools can be provided to help develop applications and scripts.

Selecting an IDE

A common question is, *""What is the" "best"" IDE for doing Python development?""* The short answer to this question is that the IDE choice doesn't matter very much. The number of development environments that support Python is vast and they are all very easy to use. The long answer requires a conversation about what attributes would rank an IDE as being the *best*.

The Spyder IDE is part of the Anaconda distribution. This makes it readily accessible to developers who've downloaded Anaconda. The IDLE editor is part of the Python distribution, and provides a simple environment for using Python and building scripts. PyCharm has a commercial license as well as a community edition, it provides a large number of features, and was used to prepare all the examples in this book.

The author makes use of having both an editor, an integrated Python prompt, and unit test results all readily available. PyCharm works well with the `conda` environments, avoiding confusion over what packages are installed.

A search on the internet will provide a long list of other tools. See the IDE Python wiki page for numerous alternatives (`https://wiki.python.org/moin/IntegratedDevelopmentEnvironments`).

Consistency and style

All of the examples in the book were prepared using the `black` tool to provide consistent formatting. Some additional manual adjustments were made to keep code within the narrow sizes of printed material.

A common alternative to using `black` is to use `pylint` to identify formatting problems. These can then be corrected. In addition to detailed analysis of code quality, the `pylint` tool offers a numeric quality score. For this book, some `pylint` rules needed to be disabled. For example, the modules often have imports that are not in the preferred order; some modules also have imports that are relevant to `doctest` examples, and appear to be unused; some examples use global variables; and some class definitions are mere skeletons without appropriate method definitions.

Using `pylint` to locate potential problems is essential. It's often helpful to silence `pylint` warnings. In the following example, we need to silence a `pylint` warning about the `test_list` variable name being invalid as a global variable:

```
# pylint: disable=invalid-name
test_list = """
   >>> f = [1, 1, 2, 3]
    >>> f += [f[-1] + f[-2]]
    >>> f
    [1, 1, 2, 3, 5]
   """

if __name__ == "__main__":
    import doctest
    __test__ = {name: value
        for name, value in locals().items()
            if name.startswith("test_")}
    doctest.testmod(verbose=False)
```

Besides helping enforce a consistent style, the `pylint` warnings are helpful for identifying spelling mistakes and a list of common errors. For example, the instance variable is commonly `self`. An accidental spelling error of `sefl` will be found by `pylint`.

Type hints and the mypy program

Python 3 permits the use of type hints. The hints are present in assignment statements, function, and class definitions. They're not used directly by Python when the program runs. Instead, they're used by external tools to examine the code for improper use of types, variables, and functions. Here's a simple function with type hints:

```
def F(n: int) -> int:
    if n in (0, 1):
        return 1
    else:
        return F(n-1) + F(n-2)

print("Good Use", F(8))
print("Bad Use", F(355/113))
```

When we run the `mypy` program, we'll see an error such as the following:

```
Chapter_1/ch01_ex3.py:23: error: Argument 1 to "F" has incompatible type "float"; expect
```

This message informs us of the location of the error: the file is `Chapter_1/ch01_ex3.py`, which is the 23rd line of the file. The details tell us that the function, `F`, has an improper argument value. This kind of problem can be difficult to see. In some cases, unit tests might not cover this case very well, and it's possible for a program to harbor subtle bugs because data of an improper type might be used.

Performance – the timeit module

We'll make use of the `timeit` module to compare the actual performance of different object-oriented designs and Python constructs. We'll focus on the `timeit()` function in this module. This function creates a `Timer` object that's used to measure the execution of a given block of code. We can also provide some preparatory code that creates an environment. The return value from this function is the time required to run the given block of code.

The default count is 100,000. This provides a meaningful time that averages out other OS-level activity on the computer doing the measurement. For complex or long-running statements, a lower count may be prudent.

Here's a simple interaction with `timeit`:

```
>>> timeit.timeit("obj.method()",
... """
... class SomeClass:
...     def method(self):
...         pass
... obj= SomeClass()
... """)
0.1980541350058047
```

The code to be measured is `obj.method()`. It is provided to `timeit()` as a string. The setup code block is the class definition and object construction. This code block, too, is provided as a string. It's important to note that everything required by the statement must be in the setup. This includes all imports, as well as all variable definitions and object creation.

This example showed that 100,000 method calls that do nothing costs 0.198 seconds.

Testing – unittest and doctest

Unit testing is absolutely essential.

If there's no automated test to show a particular element functionality, then the feature doesn't really exist. Put another way, it's not done until there's a test that shows that it's done.

We'll touch, tangentially, on testing. If we delved into testing each object-oriented design feature, the book would be twice as long as it is. Omitting the details of testing has the disadvantage of making good unit tests seem optional. They're emphatically not optional.

> *Unit testing is essential.*
>
> *When in doubt, design the tests first. Fit the code to the test cases.*

Python offers two built-in testing frameworks. Most applications and libraries will make use of both. One general wrapper for testing is the `unittest` module. In addition, many public API docstrings will have examples that can be found and used by the `doctest` module. Also, `unittest` can incorporate `doctest`.

The `pytest` tool can locate test cases and execute them. This is a very useful tool, but must be installed separately from the rest of Python.

One lofty ideal is that every class and function has at least a unit test. The important, visible classes and functions will often also have `doctest`. There are other lofty ideals: 100% code coverage; 100% logic path coverage, and so on.

Pragmatically, some classes don't need testing. A class that extends `typing.NamedTuple`, for example, doesn't really need a sophisticated unit test. It's important to test the unique features of a class you've written and not the features inherited from the standard library.

Generally, we want to develop the test cases first, and then write code that fits the test cases. The test cases formalize the API for the code. This book will reveal numerous ways to write code that has the same interface. Once we've

defined an interface, there are still numerous candidate implementations that fit the interface. One set of tests will apply to several different object-oriented designs.

One general approach to using the `unittest` and `pytest` tools is to create at least three parallel directories for your project:

- `myproject`: This directory is the final package that will be installed in `lib/site-packages` for your package or application. It has an `__init__.py` file. We'll put our files in here for each module.
- `tests`: This directory has the test scripts. In some cases, the scripts will parallel the modules. In some cases, the scripts may be larger and more complex than the modules themselves.
- `docs`: This has other documentation. We'll touch on this in the next section, as well as a chapter in part three.

In some cases, we'll want to run the same test suite on multiple candidate classes so that we can be sure each candidate works. There's no point in doing `timeit` comparisons on code that doesn't actually work.

Documentation – sphinx and RST markup

All Python code should have docstrings at the module, class and method level. Not every single method requires a docstring. Some method names are really well chosen, and little more needs to be said. Most times, however, documentation is essential for clarity.

Python documentation is often written using the **reStructuredText** (**RST**) markup.

Throughout the code examples in the book, however, we'll omit docstrings. The omission keeps the book to a reasonable size. This gap has the disadvantage of making docstrings seem optional. They're emphatically not optional.

> TIP: *This point is so important, we'll emphasize it again:* ***docstrings are essential.***

The docstring material is used three ways by Python:

- The internal `help()` function displays the docstrings.
- The `doctest` tool can find examples in docstrings and run them as test cases.
- External tools, such as `sphinx` and `pydoc`, can produce elegant documentation extracts from these strings.

Because of the relative simplicity of RST, it's quite easy to write good docstrings. We'll look at documentation and the expected markup in detail in `Chapter 18`, *Coping with the Command Line*. For now, however, we'll provide a quick example of what a docstring might look like:

```
def factorial(n: int) -> int:
    """
    Compute n! recursively.

    :param n: an integer >= 0
    :returns: n!

    Because of Python's stack limitation, this won't compute a value larger than about 1

    >>> factorial(5)
```

```
    120
    """
    if n == 0:
        return 1
    return n*factorial(n-1)
```

This shows the RST markup for the `n` parameter and the return value. It includes an additional note about limitations. It also includes a `doctest` example that can be used to validate the implementation using the `doctest` tool. The use of `:param n:` and `:return:` identifies text that will be used by the `sphinx` tool to provide proper formatting and indexing of the information.

Installing components

Most of the tools required must be added to the Python 3.7 environment. There are two approaches in common use:

- Use `pip` to install everything.
- Use `conda` to create an environment. Most of the tools described in this book are part of the Anaconda distribution.

The `pip` installation uses a single command:

```
python3 -m pip install pyyaml sqlalchemy jinja2 pytest sphinx mypy pylint black
```

This will install all of the required packages and tools in your current Python environment.

The `conda` installation creates a `conda` environment to keep the book's material separate from any other projects:

1. Install `conda`. If you have already installed Anaconda, you have the Conda tool, nothing more needs to be done. If you don't have Anaconda yet, then install `miniconda`, which is the ideal way to get started. Visit https://conda.io/miniconda.html and download the appropriate version of `conda` for your platform.
2. Use `conda` to build and activate the new environment.

3. Then upgrade `pip`. This is needed because the default `pip` installation in the Python 3.7 environment is often slightly out of date.
4. Finally, install `black`. This is required because `black` is not currently in any of the `conda` distribution channels.

Here are the commands:

```
$ conda create --name mastering python=3.7 pyyaml sqlalchemy jinja2
  pytest sphinx mypy pylint
$ conda activate mastering
$ python3 -m pip install --upgrade pip
$ python3 -m pip install black
```

The suite of tools (`pytest`, `sphinx`, `mypy`, `pylint`, and `black`) are essential for creating

high-quality, reliable Python programs. The other components, `pyyaml`, `sqlalchemy`, and `jinja2`, are helpful for building useful applications.

Summary

In this chapter, we've surveyed the game of *Blackjack*. The rules have a moderate level of complexity, providing a framework for creating a simulation. Simulation was one of the first uses for OOP and remains a rich source of programming problems that illustrate language and library strengths.

This chapter introduces the way the Python runtime uses special methods to implement the various operators. The bulk of this book will show ways to make use of the special methods names for creating objects that interact seamlessly with other Python features.

We've also looked at a number of tools that will be required to build good Python applications. This includes the IDE, the `mypy` program for checking type hints, and the `black` and `pylint` programs for getting to a consistent style. We also looked at the `timeit`, `unittest`, and `doctest` modules for doing essential performance and functional testing. For final documentation of a project, it's helpful to install `sphinx`. The installation of these extra components can be done with `pip` or `conda`. The `pip` tool is part of Python, the `conda` tool requires another download to make it available.

In the next chapter, we'll start our exploration of Python with class definition. We'll focus specifically on how objects are initialized using the `__init__()` special method.

The __init__() Method

The `__init__()` method is a profound feature of Python class definitions for two reasons. Firstly, initialization is the first big step in an object's life; every object must have its state initialized properly. The second reason is that the argument values for `__init__()` can take many forms.

Because there are so many ways to provide argument values to `__init__()`, there is a vast array of use cases for object creation. We'll take a look at several of them. We want to maximize clarity, so we need to define an initialization that characterizes the problem domain and clearly sets the state of the object.

Before we can get to the `__init__()` method, however, we need to take a look at the implicit class hierarchy in Python, glancing briefly at the class named `object`. This will set the stage for comparing its default behavior with the different kinds of behavior we want from our own classes.

In this chapter, we will take a look at different forms of initialization for simple objects (for example, playing cards). After this, we will take a look at more complex objects, such as hands, which involve collections, and players, which involve strategies and states. Throughout these examples, we'll include type hints and explain how `mypy` will examine this code to determine the correct use of objects.

In this chapter, we will cover the following topics:

- All Python objects are subclasses of a common parent, the `object` class, so we'll look at this first.
- We'll look at how the default `__init__()` method for the `object` class works.
- The first design strategy we'll look at is using a common `__init__()` method for all subclasses of a hierarchy. This can lead to using a factory function, separate from the `__init__()` method, to help initialize objects correctly.
- The second design strategy involves pushing the `__init__()` method into each individual subclass of a complex hierarchy, and how this changes the design of the classes.
- We'll look at how to create composite objects, which involves a number of

related uses of the `__init__()` methods of different classes.
- We'll also look at stateless objects, which don't need a sophisticated `__init__()` method.
- The chapter will finish with several more complex uses of class-level (or static) initialization, and how to validate values before creating an invalid object.

In the first section, we'll look at Python's superclass for all objects, the `object` class.

Technical requirements

The code files for this chapter can be found at `https://git.io/fj2U0`.

The implicit superclass – object

Each Python class definition has an implicit superclass: `object`. It's a very simple class definition that does almost nothing.

We can create instances of `object`, but we can't do much with them, because many of the special methods simply raise exceptions.

When we define our own class, `object` is the superclass. The following is an example class definition that simply extends `object` with a new name:

```
>>> class X:
>>>     pass
```

The following are some interactions with this tiny class definition:

```
>>> X.__class__
<class 'type'>
>>> X.__class__.__base__
<class 'object'>
```

We can see that a class is an object of the class named `type` and that the base class for our new class is the class named `object`. As we look at each method, we also take a look at the default behavior inherited from `object`. In some cases, the superclass special method's behavior will be exactly what we want. In other cases, we'll need to override the behavior of the special method.

The base class object __init__() method

Fundamental to the life cycle of an object are its creation, initialization, and destruction. We'll defer creation and destruction to a later chapter on more advanced special methods and focus on initialization. This will set the initial state of the object.

The superclass of all classes, `object`, has a default implementation of `__init__()` that amounts to `pass`. We aren't required to implement `__init__()`. If we don't implement it, then no instance variables will be created when the object is created. In some cases, this default behavior is acceptable.

We can add attributes to an object that's a subclass of `object`. Consider the following class, which requires two instance variables, but doesn't initialize them:

```
class Rectangle:
    def area(self) -> float:
        return self.length * self.width
```

The `Rectangle` class has a method that uses two attributes to return a value. The attributes have not been initialized anywhere in the class definition. While this is legal Python, it's a little strange to avoid specifically setting attributes. The following is an interaction with the `Rectangle` class:

```
>>> r = Rectangle()
>>> r.length, r.width = 13, 8
>>> r.area()
104
```

While this is legal, it's a potential source of deep confusion, which is a good reason to avoid it. Setting ad-hoc attribute values outside the class body in the example shown above defeats type hint checking by `mypy`, which is another reason for avoiding it.

This kind of design grants some flexibility, so there could be times when we needn't set all of the attributes in the `__init__()` method. We walk a fine line here.

An optional attribute implies a kind of subclass that's not formally declared as a proper subclass.

We're creating polymorphism in a way that could lead to confusing and inappropriate use of convoluted `if` statements. While uninitialized attributes may be useful, they could be a symptom of bad design.

The Zen of Python, by Tim Peters, available from the standard library via `import this`, offers the following advice:

"Explicit is better than implicit."

This statement has proven helpful over the years, to help keep Python programs simple and consistent. This is **Python Enhancement Proposal** (**PEP**) number 20. See https://www.python.org/dev/peps/pep-0020/ for further information.

An `__init__()` method should make instance variables explicit.

> *Pretty poor polymorphism*
> *There's a fine line between flexibility and foolishness. We may have stepped over the edge of flexible into foolish as soon as we feel the need to write the following:*

```
if 'x' in self.__dict__:
    code-to-handle-optional-attribute
```

> TIP
>
> *Or, we could see the following:*

```
try:
    self.x
except AttributeError:
    code-to-handle-optional-attribute
```

> *It's time to reconsider the API and add a common method or attribute. Refactoring is better than adding `if` statements.*

Implementing __init__() in a superclass

We initialize an object by implementing the `__init__()` method. When an object is created, Python first creates an empty object and then calls the `__init__()` method to set the state of the new object. This method generally creates the object's instance variables and performs any other one-time processing.

The following are some example definitions of a `Card` class hierarchy. We'll define a `Card` superclass and three subclasses that are variations of the basic theme of `Card`. We have two instance variables that have been set directly from argument values and two variables that have been calculated using an initialization method:

```
from typing import Tuple

class Card:

    def __init__(self, rank: str, suit: str) -> None:
        self.suit = suit
        self.rank = rank
        self.hard, self.soft = self._points()

    def _points(self) -> Tuple[int, int]:
        return int(self.rank), int(self.rank)

class AceCard(Card):

    def _points(self) -> Tuple[int, int]:
        return 1, 11

class FaceCard(Card):

    def _points(self) -> Tuple[int, int]:
        return 10, 10
```

In this example, we factored the `__init__()` method into the superclass so that a common initialization in the superclass, `Card`, applies to two subclasses, `AceCard` and `FaceCard`.

This example provides type hints for parameters of the `__init__()` method. Both the `rank` and `suit` parameters are expected to have values of the `str` type. The result of the `__init__()` method is always `None`, since it never returns a value. These

hints can be checked by the `mypy` tool to ensure that the class is used properly.

This shows a common polymorphic design. Each subclass provides a unique implementation of the `_points()` method. The various `_points()` methods all return a two-tuple with the different ways to evaluate a card. All the subclasses have identical signatures – they have the same methods and attributes. Objects of these three subclasses can be used interchangeably in an application.

The leading _ in the name is a suggestion to someone reading the class that the `_points()` method is an implementation detail, subject to change in a future implementation. This can help to reveal which methods are part of a public interface and which are details that aren't intended for general use by other classes.

If we simply use characters for suits, we will be able to create the `Card` instances, as shown in the following code snippet:

```
cards = [AceCard('A', '♠'), Card('2','♠'), FaceCard('J','♠'),]
```

We enumerated the class, rank, and suit for several cards in a list. In the long run, we'll need a much smarter factory function to build `Card` instances; enumerating all 52 cards this way is tedious and error-prone. Before we get to factory functions, we will take a look at a number of other issues.

Creating enumerated constants

We can define classes for the suits of our cards. The suits of playing cards are an example of a type with a domain that can be exhaustively enumerated. Some other types with very small domains of values include the `None` type, where there's only one value, and the `bool` type, which has only two values.

The suit of a playing card could be thought of as an immutable object: the state should not be changed. Python has one simple formal mechanism for defining an object as immutable. We'll look at techniques to assure immutability in Chapter 4, *Attribute Access, Properties, and Descriptors*. While it might make sense for the attributes of a suit to be immutable, the extra effort has no tangible benefit.

The following is a class that we'll use to build four manifest constants:

```
from enum import Enum


class Suit(str, Enum):
    Club = "♣"
    Diamond = "♦"
    Heart = "♥"
    Spade = "♠"
```

This class has two parent classes. Each of the four values of the `Suit` class is both a string as well as an `Enum` instance. Each string value is only a single Unicode character. The enumerated values must be qualified by the class name, assuring that there will be no collisions with other objects.

Here's one of the enumerated constants built by this class:

```
>>> Suit.Club
<Suit.Club: '♣'>
```

The representation of an `Enum` instance shows the name within the `Enum` class, as well as the value assigned by the other parent class. To see only the value, use an expression such as `Suit.Heart.value`.

We can now create `cards`, as shown in the following code snippet:

```
cards = [AceCard('A', Suit.Spade), Card('2', Suit.Spade), FaceCard('Q', Suit.Spade),]
```

For an example this small, this class isn't a huge improvement on single character suit codes. It is very handy to have the explicit enumeration of the domain of values. An expression such as `list(Suit)` will provide all of the available objects.

We do have to acknowledge that these objects aren't technically immutable. It's possible to assign additional attributes to the `Suit` objects. While additional attributes can be added, the `value` attribute cannot be changed. The following example shows the exception raised:

```
>>> Suit.Heart.value = 'H'
Traceback (most recent call last):
    File "<doctest __main__.__test__.test_suit_value[1]>", line 1, in <module>
        Suit.Heart.value = 'H'
    File "/Users/slott/miniconda3/envs/py37/lib/python3.7/types.py", line 175, in __set_
        raise AttributeError("can't set attribute")
AttributeError: can't set attribute
```

TIP

The irrelevance of immutability

Immutability can become an attractive nuisance. It's sometimes justified by a mythical malicious programmer who modifies the constant value in their application. As a design consideration, this is often silly. A mythical malicious programmer can't be stopped by creating immutable objects. A malicious programmer would have access to the Python source and be able to tweak it just as easily as they could write poorly-crafted code to modify a constant.

In `Chapter 4`, *Attribute Access, Properties, and Descriptors*, we'll show you some ways to provide suitable diagnostic information for a buggy program that's attempting to mutate objects intended to be immutable.

Leveraging __init__() via a factory function

We can build a complete deck of cards via a factory function. This beats enumerating all 52 cards. In Python, there are two common approaches to factories, as follows:

- We define a function that creates objects of the required classes.
- We define a class that has methods for creating objects. This is the **Factory** design pattern, as described in books on object-oriented design patterns. In languages such as Java, a **factory** class hierarchy is required because the language doesn't support standalone functions.

In Python, a class isn't *required* to create an object factory, but this can be a good idea when there are related factories or factories that are complex. One of the strengths of Python is that we're not forced to use a class hierarchy when a simple function might do just as well.

> *While this is a book about object-oriented programming, a function really is fine. It's common, idiomatic Python.*

We can always rewrite a function to be a proper callable object if the need arises. From a callable object, we can refactor it into a class hierarchy for our factories. We'll look at callable objects in `Chapter 6`, *Using Callables and Contexts*.

The advantage of class definitions is code reuse via inheritance. The purpose of a factory class is to encapsulate the complexities of object construction in a way that's extensible. If we have a factory class, we can add subclasses when extending the target class hierarchy. This can give us polymorphic factory classes; different factory class definitions can have the same method signatures and can be used interchangeably.

If the alternative factory definitions don't actually reuse any code, then a class hierarchy won't be as helpful in Python. We can simply use functions that have the same signatures.

The following is a factory function for our various `Card` subclasses:

```
def card(rank: int, suit: Suit) -> Card:
    if rank == 1:
        return AceCard("A", suit)
    elif 2 <= rank < 11:
        return Card(str(rank), suit)
    elif 11 <= rank < 14:
        name = {11: "J", 12: "Q", 13: "K"}[rank]
        return FaceCard(name, suit)
    raise Exception("Design Failure")
```

This function builds a `Card` class from a numeric `rank` number and a `suit` object. The type hints clarify the expected argument values. The `-> Card` hint describes the result of this function, showing that it will create a `Card` object. We can now build `Card` instances more simply. We've encapsulated the construction issues into a single factory function, allowing an application to be built without knowing precisely how the class hierarchy and polymorphic design works.

The following is an example of how we can build a deck with this factory function:

```
deck = [card(rank, suit)
    for rank in range(1,14)
        for suit in iter(Suit)]
```

This enumerates all the ranks and suits to create a complete deck of 52 cards. This works nicely, because the `Enum` subclasses will iterate over the list of enumerated values.

We do not need to use `iter(Suit)`. We can use `Suit` in the preceding generator, and it will work nicely. While the `for suit in Suit` form will work, `mypy` will signal errors. Using `list(Suit)` or `iter(Suit)` will mute the errors by making the intent clear.

Faulty factory design and the vague else clause

Note the structure of the `if` statement in the `card()` function. We did not use a catch-all `else` clause to do any processing; we merely raised an exception. The use of a catch-all `else` clause is subject to debate.

On the one hand, it can be argued that the condition that belongs in an `else` clause should never be left unstated because it may hide subtle design errors. On the other hand, some `else` clause conditions are truly obvious.

It's important to avoid a vague `else` clause.

Consider the following variant on this factory function definition:

```
def card2(rank: int, suit: Suit) -> Card:
    if rank == 1:
        return AceCard("A", suit)
    elif 2 <= rank < 11:
        return Card(str(rank), suit)
    else:
        name = {11: "J", 12: "Q", 13: "K"}[rank]
        return FaceCard(name, suit)
```

While this kind of code is common, it's not *perfectly* clear what condition applies to the `else:` clause.

The following looks like it might build a valid deck:

```
deck2 = [card2(rank, suit) for rank in range(13) for suit in iter(Suit)]
```

This doesn't work. But the error is an obscure `KeyError` when trying to build a `FaceCard` instance.

What if the `if` conditions were more complex? While some programmers will understand this `if` statement at a glance, others will struggle to determine whether all of the cases are properly exclusive.

We should not force the reader to deduce a complex condition for an `else` clause.

Either the condition should be obvious to the newest of noobz, or it should be explicit.

> *Catch-all else should be used rarely. Use it only when the condition is obvious. When in doubt, be explicit and use `else` to raise an exception. Avoid the vague `else` clause.*

Simplicity and consistency using elif sequences

The factory function, `card()`, is a mixture of two very common Factory design patterns:

- An `if-elif` sequence
- A mapping

For the sake of simplicity, it can be better to focus on just one of these techniques rather than on both.

We can always replace a mapping with `elif` conditions. (Yes, always. The reverse is not true though; transforming `elif` conditions to a mapping can be challenging.)

The following is a `Card` factory without a mapping:

```
def card3(rank: int, suit: Suit) -> Card:
    if rank == 1:
        return AceCard("A", suit)
    elif 2 <= rank < 11:
        return Card(str(rank), suit)
    elif rank == 11:
        return FaceCard("J", suit)
    elif rank == 12:
        return FaceCard("Q", suit)
    elif rank == 13:
        return FaceCard("K", suit)
    else:
        raise Exception("Rank out of range")
```

We rewrote the `card()` factory function. The mapping was transformed into additional `elif` clauses. This function has the advantage that it is more consistent than the previous version.

Simplicity using mapping and class objects

In some cases, we can use a mapping instead of a chain of `elif` conditions. It's possible to find conditions that are so complex that a chain of `elif` conditions is the only sensible way to express them. For simple cases, however, a mapping often works better and can be easy to read.

Since `class` is a first-class object, we can easily map from the `rank` parameter to the class that must be constructed.

The following is a `Card` factory that uses only a mapping:

```
def card4(rank: int, suit: Suit) -> Card:
    class_ = {1: AceCard, 11: FaceCard, 12: FaceCard,
        13: FaceCard}.get(rank, Card)
    return class_(str(rank), suit)
```

We've mapped the `rank` object to a class. Then, we applied the class to the `rank` and `suit` values to build the final `Card` instance.

The `card4()` function, however, has a serious deficiency. It lacks the translation from `1` to `A` and `13` to `K` that we had in previous versions. When we try to add that feature, we run into a problem.

We need to change the mapping to provide both a `Card` subclass as well as the string version of the `rank` object. How can we create this two-part mapping? There are four common solutions:

- We can do two parallel mappings. We don't suggest this, but we'll show it to emphasize what's undesirable about it.
- We can map to a two-tuple. This also has some disadvantages.
- We can map to a `partial ()` function. The `partial()` function is a feature of the `functools` module. This won't work out perfectly, and we'll use a lambda object to achieve the same goal.
- We can also consider modifying our class definition to fit more readily with this kind of mapping. We'll look at this alternative in the next section, on

pushing `__init__()` into subclass definitions.

We'll look at each of these with a concrete example.

Two parallel mappings

The following is the essence of the two-parallel mappings solution:

```
def card5(rank: int, suit: Suit) -> Card:
    class_ = {1: AceCard, 11: FaceCard, 12: FaceCard,
        13: FaceCard}.get(rank, Card)
    rank_str = {1: "A", 11: "J", 12: "Q",
        13: "K"}.get(rank, str(rank))
    return class_(rank_str, suit)
```

This is not desirable. It involves a repetition of the sequence of the mapping keys `1`, `11`, `12`, and `13`. Repetition is bad, because parallel structures never seem to stay that way after the software has been updated or revised.

Don't use parallel structures
Two parallel structures should be replaced with tuples or some kind of proper collection.

Mapping to a tuple of values

The following is the essence of how mapping is done to a two-tuple:

```
def card6(rank: int, suit: Suit) -> Card:
    class_, rank_str = {
        1: (AceCard, "A"),
        11: (FaceCard, "J"),
        12: (FaceCard, "Q"),
        13: (FaceCard, "K")
    }.get(
        rank, (Card, str(rank))
    )
    return class_(rank_str, suit)
```

This is a reasonably pleasant design because it uses a simple mapping. It's not much code to handle special cases of playing cards. We will see how it could be modified or expanded if we needed to alter the `Card` class hierarchy to add additional subclasses of `Card`.

It does feel odd to map a `rank` value to a `class` object and one of the two arguments to that class initializer. It seems more sensible to map the rank to a simple class or function object without the clutter of providing some (but not all) of the arguments.

The partial function solution

In the previous example, we mapped the rank to a two-tuple of the class and one of the arguments for creating an instance. We can combine `class` and `rank` into a partial function. This is a function that has some argument values and is waiting for the final argument value. In many cases, we can use the `partial()` function from the `functools` library to create a partial function combining the `class` object with the `rank` argument.

The `partial()` function is not designed to create objects; using it like this will raise an exception. Instead of using the partial function, we can create a `lambda` object instead. For example, the `lambda suit: AceCard("A", suit)` expression is a function that is waiting for the suit value to create a complete `Card`.

The following is a mapping from `rank` to a `lambda` object that can be used to construct `Card` objects:

```
def card7(rank: int, suit: Suit) -> Card:
    class_rank = {
        1: lambda suit: AceCard("A", suit),
        11: lambda suit: FaceCard("J", suit),
        12: lambda suit: FaceCard("Q", suit),
        13: lambda suit: FaceCard("K", suit),
    }.get(
        rank, lambda suit: Card(str(rank), suit)
    )
    return class_rank(suit)
```

The mapping associates a `rank` object with a `lambda` object that contains a class and a string. The `lambda` object is a function that is then applied to the `suit` object to create the final `Card` instance.

This use of the `partial()` function is a common technique for functional programming. It is one way to achieve a kind of polymorphism so that several different functions can be used in a similar way.

In general, however, partial functions aren't helpful for most object-oriented programming. When building complex objects, it is common to define methods that accept arguments incrementally. Instead of using `rank` to create a partial function, a more object-oriented approach is to use separate methods to set `rank`

and `suit`.

Fluent APIs for factories

In Python, a fluent interface is built by creating methods that return the `self` instance variable. Each method can set some of the object's state. By returning `self`, the functions can be chained together.

We might have `X().a().b()` in an object notation. We can think of it as $b(a(X))$. The `x.a()` function is a kind of `partial()` function that's waiting for `b()`. We can think of `X().a()` as if it were an object with another function as its argument value, $a_X(b)$.

The idea here is that Python offers us two alternatives for initializing state. We can either set all of the values in `__init__()`, or we can set values in a number of separate methods. In the following example, we'll make the setting of the `rank` object a fluent method that returns `self`. Setting the `suit` object will actually create the `Card` instance. The following is a fluent `Card` factory class. An instance must use the two method functions in the required order:

```
class CardFactory:

    def rank(self, rank: int) -> "CardFactory":
        self.class_, self.rank_str = {
            1: (AceCard, "A"),
            11: (FaceCard, "J"),
            12: (FaceCard, "Q"),
            13: (FaceCard, "K"),
        }.get(
            rank, (Card, str(rank))
        )
        return self

    def suit(self, suit: Suit) -> Card:
        return self.class_(self.rank_str, suit)
```

The `rank()` method updates the state of the constructor, and the `suit()` method actually creates the final `Card` object. The type hint for the `rank()` function shows the function returning a `CardFactory` object. Because the class is not fully defined, the name isn't known, and a quoted string has to be used. The `mypy` tool will resolve the string type name to create a circular reference from the class to itself.

This factory class can be used as follows:

```
card8 = CardFactory()
deck8 = [card8.rank(r + 1).suit(s) for r in range(13) for s in Suit]
```

First, we create a factory instance, then we use that instance to create the `Card` instances. This doesn't materially change how `__init__()` itself works in the `Card` class hierarchy. It does, however, change the way that our client application creates objects.

Implementing __init__() in each subclass

As we look at the factory functions for creating `Card` objects, there are some alternative designs for the `Card` class. We might want to refactor the conversion of the rank number so that it is the responsibility of the `Card` class itself. This pushes the initialization down into each subclass.

This often requires some common initialization of a superclass as well as subclass-specific initialization. We need to follow the **Don't Repeat Yourself** (**DRY**) principle to keep the code from getting cloned into each of the subclasses.

This version of the `Card3` class has an initializer at the superclass level that is used by each subclass, as shown in the following code snippet:

```
class Card3:

    def __init__(
            self, rank: str, suit: Suit, hard: int, soft: int
    ) -> None:
        self.rank = rank
        self.suit = suit
        self.hard = hard
        self.soft = soft

class NumberCard3(Card3):

    def __init__(self, rank: int, suit: Suit) -> None:
        super().__init__(str(rank), suit, rank, rank)

class AceCard3(Card3):

    def __init__(self, rank: int, suit: Suit) -> None:
        super().__init__("A", suit, 1, 11)

class FaceCard3(Card3):

    def __init__(self, rank: int, suit: Suit) -> None:
        rank_str = {11: "J", 12: "Q", 13: "K"}[rank]
        super().__init__(rank_str, suit, 10, 10)
```

We've provided `__init__()` at both the subclass and superclass level. Each subclass uses the `super()` function to locate the superclass version of `__init__()`. The superclass version has a number of parameters that can be omitted from the

subclass initializers.

This has the small advantage that it simplifies our factory function, as shown in the following code snippet:

```
def card10(rank: int, suit: Suit) -> Card3:
    if rank == 1:
        return AceCard3(rank, suit)
    elif 2 <= rank < 11:
        return NumberCard3(rank, suit)
    elif 11 <= rank < 14:
        return FaceCard3(rank, suit)
    else:
        raise Exception("Rank out of range")
```

We can see from this variation that we've created rather complex `__init__()` methods for a relatively minor improvement in the simplicity of a factory function. This is a common trade-off. The complexity cannot be removed; it can only be encapsulated. The real question is how should responsibility be allocated for this complexity?

> TIP
>
> *Factory functions encapsulate complexity*
> *There's a trade-off that occurs between sophisticated `__init__()` methods and factory functions. It's often better to push complex constructors into factory functions. A factory function helps separate construction and initial state-from-state change or other processing concerns.*

Composite objects

A composite object can also be called a **container**. We'll look at a simple composite object: a deck of individual cards. This is a basic collection. Indeed, it's so basic that we can, without too much struggle, use a simple `list` object as a deck.

Before designing a new class, we need to ask this question: is using a simple `list` object appropriate?

We can use `random.shuffle()` to shuffle the deck and `deck.pop()` to deal cards into a player's `Hand`.

Some programmers rush to define new classes as if using a built-in class violates some object-oriented design principle. Avoiding a new class leaves us with the following code snippet:

```
>>> d = [card(r + 1, s) for r in range(13) for s in iter(Suit)]
>>> random.shuffle(d)
>>> hand = [d.pop(), d.pop()]
>>> hand
[FaceCard(suit=<Suit.Club: '♣'>, rank='J'), Card(suit=<Suit.Spade: '♠'>, rank='2')]
```

If it's that simple, why write a new class?

Defining a class has the advantage of creating a simplified, implementation-free interface to the object. In the case of the `list` example shown in the preceding code, it's not clear how much simpler a `Deck` class would be.

The deck has two use cases. A class definition doesn't seem to simplify things very much. It does have the advantage of concealing the implementation's details. In this example, the details are so trivial that exposing them has little cost.

We're focused primarily on the `__init__()` method in this chapter, so we'll look at some designs to create and initialize a collection. To design a collection of objects, we have the following three general design strategies:

- **Wrap**: This design pattern surrounds an existing collection definition with a simplified interface. This is an example of the more general **Facade** design pattern.
- **Extend**: This design pattern starts with an existing collection class and extends it to add features.
- **Invent**: This is designed from scratch. We'll look at this in `Chapter 7`, *Creating Containers and Collections*.

These three concepts are central to object-oriented design. Because Python has so many features built into the language, we must always make this choice when designing a class.

Wrapping a collection class

The following is a wrapper design that contains an internal collection:

```
class Deck:
    def __init__(self) -> None:
        self._cards = [card(r + 1, s)
            for r in range(13) for s in iter(Suit)]
        random.shuffle(self._cards)

    def pop(self) -> Card:
        return self._cards.pop()
```

We've defined `Deck` so that the internal collection is a `list` object. The `pop()` method of `Deck` simply delegates to the wrapped `list` object.

We can then create a `Hand` instance with the following type of code:

```
d = Deck()
hand = [d.pop(), d.pop()]
```

Generally, a **Facade** design pattern or **wrapper** class contains methods that delegate the work to the underlying implementation class. This delegation can become wordy when a lot of features are provided. For a sophisticated collection, we may wind up delegating a large number of methods to the wrapped object.

Extending a collection class

An alternative to wrapping is to extend a built-in class. By doing this, we have the advantage of not having to reimplement the `pop()` method; we can simply inherit it.

The `pop()` method has an advantage in that it creates a class without writing too much code. In this example, extending the `list` class has the disadvantage that this provides many more functions than we truly need.

The following is a definition of `Deck2` that extends the built-in `list` object:

```
class Deck2(list):

    def __init__(self) -> None:
        super().__init__(
            card(r + 1, s)
                for r in range(13) for s in iter(Suit))
        random.shuffle(self)
```

In this case, we've initialized the list with `Card` instances. `super().__init__()` reaches up to the superclass initialization to populate our `list` object with an initial single deck of cards. After seeding the list, the initializer then shuffles the cards. The `pop()` method is directly inherited from `list` and works perfectly. Other methods inherited from the `list` class will also work.

While simpler, this exposes methods such as `delete()` and `remove()`. If these additional features are undesirable, a wrapped object might be a better idea.

More requirements and another design

In a casino, cards are often dealt from a shoe that has half a dozen decks of cards all mingled together. This additional complexity suggests that we need to build our own implementation of `Deck` and not simply use the `list` class directly. Additionally, a casino shoe is not dealt fully. Instead, a marker card is inserted. Because of the marker, some cards are effectively set aside and not used for play. These cards are said to be *burned*.

The following `Deck3` class definition contains multiple sets of 52-card decks:

```
class Deck3(list):

    def __init__(self, decks: int = 1) -> None:
        super().__init__()
        for i in range(decks):
            self.extend(
                card(r + 1, s)
                for r in range(13) for s in iter(Suit)
            )
        random.shuffle(self)
        burn = random.randint(1, 52)
        for i in range(burn):
            self.pop()
```

Here, we used the `__init__()` method of the superclass to build an empty collection. Then, we used `self.extend()` to extend this collection with multiple 52-card decks. This populates the shoe. We could also use `super().extend()`, since we did not provide an overriding implementation in this class.

We could also carry out the entire task via `super().__init__()` using a more deeply nested generator expression, as shown in the following code snippet:

```
super().__init__(
    card(r + 1, s)
        for r in range(13)
            for s in iter(Suit)
                for d in range(decks)
    )
```

This class provides us with a collection of `Card` instances that we can use to emulate casino blackjack as dealt from a shoe.

Complex composite objects

The following is an example of a *Blackjack* `Hand` description that might be suitable for emulating play strategies:

```
class Hand:

    def __init__(self, dealer_card: Card) -> None:
        self.dealer_card: Card = dealer_card
        self.cards: List[Card] = []

    def hard_total(self) -> int:
        return sum(c.hard for c in self.cards)

    def soft_total(self) -> int:
        return sum(c.soft for c in self.cards)

    def __repr__(self) -> str:
        return f"{self.__class__.__name__} {self.dealer_card} {self.cards}"
```

In this example, we have a `self.dealer_card` instance variable based on a parameter of the `__init__()` method. The `self.cards` instance variable, however, is not based on any parameter. This kind of initialization creates an empty collection. Note that the assignment to the `self.cards` variable requires a type hint to inform `mypy` of the expected contents of the `self.cards` collection.

To create an instance of `Hand`, we can use the following code:

```
>>> d = Deck()
>>> h = Hand(d.pop())
>>> h.cards.append(d.pop())
>>> h.cards.append(d.pop())
```

This has the disadvantage of consisting of a long-winded sequence of statements to build an instance of a `Hand` object. It can become difficult to serialize the `Hand` object and rebuild it with an initialization as complex as this. Even if we were to create an explicit `append()` method in this class, it would still take multiple steps to initialize the collection.

The definition of the `__repr__()` method illustrates this problem. We can't provide a simple string representation that would rebuild the object. The typical use of `__repr__()` is to create a Pythonic view of the object's state, but, with such a complex initialization, there's no simple expression to represent it.

We could try to create a fluent interface, but that wouldn't really simplify things; it would merely mean a change in the syntax of the way that a `Hand` object is built. A fluent interface still leads to multiple method evaluations. When we take a look at the serialization of objects in part 2, *Persistence and Serialization*, we'd like an interface that's a single class-level function; ideally the class constructor. We'll look at this in depth in `Chapter 10`, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML*.

You should also note that the hard total and soft total method functions shown here don't fully follow the rules of *Blackjack*. We'll return to this issue in `Chapter 3`, *Integrating Seamlessly – Basic Special Methods*.

Complete composite object initialization

Ideally, the `__init__()` initializer method will create a complete instance of an object. This is a bit more complex when creating a complete instance of a container that contains an internal collection of other objects. It'll be helpful if we can build this composite in a single step.

It's common to have both a method to incrementally accrete items, as well as the initializer special method, which can load all of the items in one step.

For example, we might have a class such as the following code snippet:

```
class Hand2:

    def __init__(self, dealer_card: Card, *cards: Card) -> None:
        self.dealer_card = dealer_card
        self.cards = list(cards)

    def card_append(self, card: Card) -> None:
        self.cards.append(card)

    def hard_total(self) -> int:
        return sum(c.hard for c in self.cards)

    def soft_total(self) -> int:
        return sum(c.soft for c in self.cards)

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}({self.dealer_card!r}, *{self.cards})"
```

This initialization sets all of the instance variables in a single step. The other methods are copies from the previous class definition. The first positional argument value is assigned to the `dealer_card` parameter. The use of `*` with the `cards` parameter means that all of the remaining positional argument values are collected into a tuple and assigned to the `cards` parameter.

We can build a `Hand2` object in two ways. This first example loads one card at a time into a `Hand2` object:

```
d = Deck()
P = Hand2(d.pop())
p.cards.append(d.pop())
p.cards.append(d.pop())
```

This second example uses the `*cards` parameter to load a sequence of the `Cards` class in a single step:

```
d = Deck()
h = Hand2(d.pop(), d.pop(), d.pop())
```

For unit testing, it's often helpful to build a composite object in a single statement in this way. More importantly, some of the serialization techniques from the next part will benefit from a way of building a composite object in a single, simple evaluation.

Stateless objects without __init__()

The following is an example of a degenerate class that doesn't need an `__init__()` method. It's a common design pattern for **Strategy** objects. A Strategy object is plugged into some kind of master or owner object to implement an algorithm or decision. The Strategy object often depends on data in the master object; the Strategy object may not have any data of its own. We often design strategy classes to follow the Flyweight design pattern so we can avoid internal storage in the strategy instance. All values can be provided to a Strategy object as method argument values. In some cases, a strategy object can be stateless; in this instance, it is more a collection of method functions than anything else.

In the following examples, we'll show both stateless and stateful strategy class definitions. We'll start with the strategy for making some of the player decisions based on the state of the `Hand` object.

In this case, we're providing the gameplay decisions for a `Player` instance. The following is an example of a (dumb) strategy to pick cards and decline other bets:

```
class GameStrategy:

    def insurance(self, hand: Hand) -> bool:
        return False

    def split(self, hand: Hand) -> bool:
        return False

    def double(self, hand: Hand) -> bool:
        return False

    def hit(self, hand: Hand) -> bool:
        return sum(c.hard for c in hand.cards) <= 17
```

Each method requires the current `Hand` object as an argument value. The decisions are based on the available information; that is, on the dealer's cards and the player's cards. The result of each decision is shown in the type hints as a Boolean value. Each method returns `True` if the player elects to perform the action.

We can build a single instance of this strategy for use by various `Player` instances, as shown in the following code snippet:

```
dumb = GameStrategy()
```

We can imagine creating a family of related strategy classes, each one using different rules for the various decisions a player is offered in *Blackjack*.

Some additional class definitions

As noted previously, a player has two strategies: one for betting and one for playing their hand. Each `Player` instance has a sequence of interactions with a larger simulation engine. We'll call the larger engine the `Table` class.

The `Table` class requires the following sequence of events by the `Player` instances:

1. The player must place an initial, or *ante*, bet based on the betting strategy.
2. The player will then receive a hand of cards.
3. If the hand is splittable, the player must decide whether to split it or not based on their game strategy. This can create additional `Hand` instances. In some casinos, the additional hands are also splittable.
4. For each `Hand` instance, the player must decide to hit, double, or stand based on their game strategy.
5. The player will then receive payouts, and they must update their betting strategy based on their wins and losses.

From this, we can see that the `Table` class has a number of API methods to receive a bet, create a `Hand` object, offer a split, resolve each hand, and pay off the bets. This is a large object that tracks the state of play with a collection of `Players`.

The following is the beginning of a `Table` class, which handles the bets and cards:

```
class Table:

    def __init__(self) -> None:
        self.deck = Deck()

    def place_bet(self, amount: int) -> None:
        print("Bet", amount)

    def get_hand(self) -> Hand2:
        try:
            self.hand = Hand2(self.deck.pop(),
                self.deck.pop(), self.deck.pop())
            self.hole_card = self.deck.pop()
        except IndexError:
            # Out of cards: need to shuffle and try again.
            self.deck = Deck()
            return self.get_hand()
        print("Deal", self.hand)
        return self.hand

    def can_insure(self, hand: Hand) -> bool:
```

```
        return hand.dealer_card.insure
```

The `Table` class is used by the `Player` class to accept a bet, create a `Hand` object, and determine whether the insurance bet is in play for this hand. Additional methods can be used by the `Player` class to get cards and determine the payout.

The exception handling shown in `get_hand()` is not a precise model of casino play. A may lead to minor statistical inaccuracies. A more accurate simulation requires the development of a deck that reshuffles itself when empty instead of raising an exception.

In order to interact properly and simulate realistic play, the `Player` class needs a betting strategy. The betting strategy is a stateful object that determines the level of the initial bet. Various betting strategies generally change a bet based on whether a game was a win or a loss.

Ideally, we'd like to have a family of `BettingStrategy` objects. Python has a module with decorators that allows us to create an abstract superclass. An informal approach to creating Strategy objects is to raise an exception for methods that *must* be implemented by a subclass.

We've defined an abstract superclass, as well as a specific subclass, as follows, to define a flat betting strategy:

```
class BettingStrategy:

    def bet(self) -> int:
        raise NotImplementedError("No bet method")

    def record_win(self) -> None:
        pass

    def record_loss(self) -> None:
        pass

class Flat(BettingStrategy):

    def bet(self) -> int:
        return 1
```

The superclass defines the methods with handy default values. The basic `bet()` method in the abstract superclass raises an exception. The subclass must override the `bet()` method. The type hints show the results of the various betting methods.

We can make use of the `abc` module to formalize an abstract superclass definition.

It would look like the following code snippet:

```
import abc
from abc import abstractmethod


class BettingStrategy2(metaclass=abc.ABCMeta):

    @abstractmethod
    def bet(self) -> int:
        return 1

    def record_win(self):
        pass

    def record_loss(self):
        pass
```

This has the advantage that it makes the creation of an instance of `BettingStrategy2`, or any subclass that failed to implement `bet()`, impossible. If we try to create an instance of this class with an unimplemented abstract method, it will raise an exception instead of creating an object.

And, yes, the abstract method has an implementation. It can be accessed via `super().bet()`. This allows a subclass to use the superclass implementation, if necessary.

Multi-strategy __init__()

We may have objects that are created from a variety of sources. For example, we might need to clone an object as part of creating a memento, or freeze an object so that it can be used as the key of a dictionary or placed into a set; this is the idea behind the `set` and `frozenset` built-in classes.

We'll look at two design patterns that offer multiple ways to build an object. One design pattern uses a complex `__init__()` method with multiple strategies for initialization. This leads to designing the `__init__()` method with a number of optional parameters. The other common design pattern involves creating multiple static or class-level methods, each with a distinct definition.

Defining an overloaded `__init__()` method can be confusing to `mypy`, because the parameters may have distinct value types. This is solved by using the `@overload` decorator to describe the different assignments of types to the `__init__()` parameters. The approach is to define each of the alternative versions of `__init__()` and decorate with `@overload`. A final version – without any decoration – defines the parameters actually used for the implementation.

The following is an example of a `Hand3` object that can be built in either of the two ways:

```
class Hand3:

    @overload
    def __init__(self, arg1: "Hand3") -> None:
        ...

    @overload
    def __init__(self, arg1: Card, arg2: Card, arg3: Card) -> None:
        ...

    def __init__(
        self,
        arg1: Union[Card, "Hand3"],
        arg2: Optional[Card] = None,
        arg3: Optional[Card] = None,
    ) -> None:
        self.dealer_card: Card
        self.cards: List[Card]

        if isinstance(arg1, Hand3) and not arg2 and not arg3:
            # Clone an existing hand
            self.dealer_card = arg1.dealer_card
```

```
            self.cards = arg1.cards
        elif (isinstance(arg1, Card)
            and isinstance(arg2, Card)
            and isinstance(arg3, Card)
        ):
            # Build a fresh, new hand.
            self.dealer_card = cast(Card, arg1)
            self.cards = [arg2, arg3]

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}({self.dealer_card!r}, *{self.cards})"
```

In the first overloaded case, a `Hand3` instance has been built from an existing `Hand3` object. In the second case, a `Hand3` object has been built from individual `Card` instances. The `@overload` decorator provides two alternative versions of the `__init__()` method. These are used by `mypy` to ensure this constructor is used properly. The undecorated version is used at runtime. It is a kind of union of the two overloaded definitions.

The `@overload` definitions are purely for `mypy` type-checking purposes. The non-overloaded definition of `__init__()` provides a hint for `arg1` as union of either a `Card` object or a `Hand3` object. The code uses the `isinstance()` function to decide which of the two types of argument values were provided. To be more robust, the `if-elif` statements should have an `else:` clause. This should raise a `ValueError` exception.

This design parallels the way a `frozenset` object can be built from individual items or an existing `set` object. We will look at creating immutable objects more in the next chapter. Creating a new `Hand3` object from an existing `Hand3` object allows us to create a memento of a `Hand3` object using a construct such as the following code snippet:

```
h = Hand3(deck.pop(), deck.pop(), deck.pop())
memento = Hand3(h)
```

We saved the `Hand` object in the `memento` variable. This can be used to compare the final with the original hand that was dealt, or we can *freeze* it for use in a set or mapping too.

More complex initialization alternatives

In order to write a multi-strategy initialization, it can seem helpful to give up on specific named parameters. This leads to trying to use the `**kw` construct to take only named arguments. This design has the advantage of being very flexible, but the disadvantage of bypassing automated type checking. It requires a great deal of documentation explaining the variant use cases.

Instead of collecting all named parameters using the `**` construct, it's often helpful to use a standalone `*` construct. When we write `def f(a: int, b: int, *, c: int)`, we're expecting two positional argument values, and the third value must be provided by name. We'd use this function as `f(1, 2, c=3)`. This provides for explicit names to cover special cases.

We can expand our initialization to also split a `Hand` object. The result of splitting a `Hand` object is simply another constructor. The following code snippet shows how the splitting of a `Hand` object might look:

```
class Hand4:

    @overload
    def __init__(self, arg1: "Hand4") -> None:
        ...

    @overload
    def __init__(self,
        arg1: "Hand4", arg2: Card, *, split: int) -> None:
        ...

    @overload
    def __init__(self,
        arg1: Card, arg2: Card, arg3: Card) -> None:
        ...

    def __init__(
        self,
        arg1: Union["Hand4", Card],
        arg2: Optional[Card] = None,
        arg3: Optional[Card] = None,
        split: Optional[int] = None,
    ) -> None:
        self.dealer_card: Card
        self.cards: List[Card]
        if isinstance(arg1, Hand4):
            # Clone an existing hand
```

```
            self.dealer_card = arg1.dealer_card
            self.cards = arg1.cards

        elif isinstance(arg1, Hand4) and isinstance(arg2, Card) and "split" is not None:
            # Split an existing hand
            self.dealer_card = arg1.dealer_card
            self.cards = [arg1.cards[split], arg2]

        elif (
            isinstance(arg1, Card)
            and isinstance(arg2, Card)
            and isinstance(arg3, Card)
        ):
            # Build a fresh, new hand from three cards
            self.dealer_card = arg1
            self.cards = [arg2, arg3]
        else:
            raise TypeError("Invalid constructor {arg1!r} {arg2!r} {arg3!r}")

    def __str__(self) -> str:
        return ", ".join(map(str, self.cards))
```

This design reflects three separate use cases:

- Creating a `Hand4` object from an existing `Hand4` object. In this case, `arg1` will have the `Hand4` type and the other arguments will have default values of `None`.
- Splitting a `Hand4` object. This requires a value for the `split` keyword argument that uses the position of the `Card` class from the original `Hand4` object. Note how `*` is inserted into the parameter list to show that the `split` value must be provided as a keyword argument value.
- Building a `Hand4` object from three `Card` instances. In this case, all three of the positional parameters will have values of the `Card` type.

The `@overload` decorator information is used by `mypy`. It provides documentation for people using this class. It has no runtime impact.

The following code snippet shows how we'd use these definitions to create and split a hand:

```
d = Deck()
h = Hand4(d.pop(), d.pop(), d.pop())
s1 = Hand4(h, d.pop(), split=0)
s2 = Hand4(h, d.pop(), split=1)
```

We created an initial `h` instance of `Hand4`, split it into two other `Hand4` instances, `s1` and `s2`, and dealt an additional `Card` class into each. The rules of *Blackjack* only allow this when the initial hand has two cards of equal rank.

While this `__init__()` method is rather complex, it has the advantage that it can

parallel the way in which `fronzenset` is created from an existing set. The disadvantage is that it needs a large docstring to explain all these variations.

Initializing with static or class-level methods

When we have multiple ways to create an object, it's sometimes more clear to use static methods to create and return instances rather than complex `__init__()` methods.

The term *static* is borrowed from other languages. Python has three kinds of binding for method functions. The default case is to bind a method to the instance; the first positional parameter is `self`, the instance variable. A method can be bound to the class; this requires the `@staticmethod` decorator. A method can also be bound to the class, but receive the class as the first positional parameter; this requires the `@classmethod` decorator.

In the case of freezing or splitting a `Hand` object, we might want to create two new static methods to freeze or split a `Hand` object. Using static methods as surrogate constructors is a tiny syntax change in construction, but it has huge advantages when organizing code.

The following is a version of `Hand` with static methods that can be used to build new instances of `Hand` from an existing `Hand` instance:

```
class Hand5:

    def __init__(self, dealer_card: Card, *cards: Card) -> None:
        self.dealer_card = dealer_card
        self.cards = list(cards)

    @staticmethod
    def freeze(other) -> "Hand5":
        hand = Hand5(other.dealer_card, *other.cards)
        return hand

    @staticmethod
    def split(other, card0, card1) -> Tuple["Hand5", "Hand5"]:
        hand0 = Hand5(other.dealer_card, other.cards[0], card0)
        hand1 = Hand5(other.dealer_card, other.cards[1], card1)
        return hand0, hand1

    def __str__(self) -> str:
        return ", ".join(map(str, self.cards))
```

The `freeze()` method freezes or creates a memento version. The `split()` method

splits a `Hand5` instance to create two new child instances of `Hand5`. The `__init__()` method builds a hand from individual `Card` instances.

This is often more readable and preserves the use of the parameter names to explain the interface. Note how the class name must be provided as a string when used as a type hint within the class definition. You will recall that the class name doesn't exist until after the execution of the `class` statement. Using strings instead of type objects permits a reference to a type that doesn't exist yet. When `mypy` evaluates the type hints, it will resolve the type objects from the strings.

The following code snippet shows how we can split a `Hand5` instance with this version of the class:

```
d = Deck()
h = Hand5(d.pop(), d.pop(), d.pop())
s1, s2 = Hand5.split(h, d.pop(), d.pop())
```

We created an initial `h` instance of `Hand5`, split it into two other hands, `s1` and `s2`, and dealt an additional `Card` class into each. The `split()` static method is much simpler than the equivalent functionality implemented via `__init__()`.

Yet more __init__() techniques

We'll take a look at a few other, more advanced `__init__()` techniques. These aren't quite so universally useful as the techniques in the previous sections.

The following is a definition for the `Player` class that uses two Strategy objects and a `table` object. This shows an unpleasant-looking `__init__()` method:

```
class Player:

    def __init__(
        self,
        table: Table,
        bet_strategy: BettingStrategy,
        game_strategy: GameStrategy
    ) -> None:
        self.bet_strategy = bet_strategy
        self.game_strategy = game_strategy
        self.table = table

    def game(self):
        self.table.place_bet(self.bet_strategy.bet())
        self.hand = self.table.get_hand()
        if self.table.can_insure(self.hand):
            if self.game_strategy.insurance(self.hand):
                self.table.insure(self.bet_strategy.bet())
        # etc. (omitted for now)
```

The `__init__()` method for `Player` seems to do little more than bookkeeping. We're simply transferring named parameters to instance variables with same name. In many cases, the `@dataclass` decorator can simplify this.

We can use this `Player` class (and related objects) as follows:

```
table = Table()
flat_bet = Flat()
dumb = GameStrategy()
p = Player(table, flat_bet, dumb)
p.game()
```

We can provide a very short and very flexible initialization by simply transferring keyword argument values directly into the internal instance variables.

The following is a way to build a `Player` class using keyword argument values:

```
class Player2(Player):
```

```
    def __init__(self, **kw) -> None:
        """Must provide table, bet_strategy, game_strategy."""
        self.bet_strategy: BettingStrategy = kw["bet_strategy"]
        self.game_strategy: GameStrategy = kw["game_strategy"]
        self.table: Table = kw["table"]

    def game(self) -> None:
        self.table.place_bet(self.bet_strategy.bet())
        self.hand = self.table.get_hand()
```

This sacrifices some readability for succinctness. Each individual instance variable now requires an explicit type hint, because the parameters don't provide any information.

Since the `__init__()` method is reduced to one line, it removes a certain level of *wordiness* from the method. This wordiness, however, is transferred to each individual object constructor expression. In effect, we provide type hints and parameter names in each of the object initialization expressions.

Here's how we must provide the required parameters, as shown in the following code snippet:

```
p2 = Player2(table=table, bet_strategy=flat_bet, game_strategy=dumb)
```

This syntax also works with the `Player` class, as shown in the preceding code. For the `Player2` class, it's a requirement. For the `Player` class, this syntax is optional.

Using the `**` construct to collect all keywords into a single variable does have a *potential* advantage. A class defined like this is easily extended. We can, with only a few specific concerns, supply additional keyword parameters to a constructor.

Here's an example of extending the preceding definition:

```
class Player2x(Player):

    def __init__(self, **kw) -> None:
        """Must provide table, bet_strategy, game_strategy."""
        self.bet_strategy: BettingStrategy = kw["bet_strategy"]
        self.game_strategy: GameStrategy = kw["game_strategy"]
        self.table: Table = kw["table"]
        self.log_name: Optional[str] = kw.get("log_name")
```

We've added a `log_name` attribute without touching the class definition. This could be used, perhaps, as part of a larger statistical analysis. The `Player2.log_name`

attribute can be used to annotate logs or other collected data. The other initialization was not changed.

We are limited in what we can add; we can only add parameters that fail to conflict with the names already in use within a class. Some knowledge of a class implementation is required to create a subclass that doesn't abuse the set of keywords already in use. Since the `**kw` parameter is opaque, we need to read it carefully. In most cases, we'd rather trust the class to work than review the implementation details. The disadvantage of this technique is the obscure parameter names, which aren't formally documented.

We can (and should) hybridize this with a mixed positional and keyword implementation, as shown in the following code snippet:

```
class Player3(Player):

    def __init__(
        self,
        table: Table,
        bet_strategy: BettingStrategy,
        game_strategy: GameStrategy,
        **extras,
    ) -> None:
        self.bet_strategy = bet_strategy
        self.game_strategy = game_strategy
        self.table = table
        self.log_name: str = extras.pop("log_name", self.__class__.__name__)
        if extras:
            raise TypeError(f"Extra arguments: {extras!r}")
```

This is more sensible than a completely open definition. We've made the required parameters positional parameters while leaving any nonrequired parameters as keywords. This clarifies the use of any extra keyword arguments given to the `__init__()` method.

The known parameter values are popped from the `extras` dictionary. After this is finished, any other parameter names represent a type error.

Initialization with type validation

Runtime type validation is rarely a sensible requirement. In a way, this might be a failure to fully understand Python. Python's type system permits numerous extensions. Runtime type checking tends to defeat this. Using `mypy` provides extensive type checking without the runtime overheads.

The notional objective behind runtime checking is to validate that all of the arguments are of a *proper* type. The issue with trying to do this is that the definition of *proper* is often far too narrow to be truly useful.

Type checking is different from checking ranges and domains within a type. Numeric range checking, for example, may be essential to prevent infinite loops at runtime.

What can create problems is trying to do something like the following in an `__init__()` method:

```
class ValidPlayer:

    def __init__(self, table, bet_strategy, game_strategy):
        assert isinstance(table, Table)
        assert isinstance(bet_strategy, BettingStrategy)
        assert isinstance(game_strategy, GameStrategy)

        self.bet_strategy = bet_strategy
        self.game_strategy = game_strategy
        self.table = table
```

The `isinstance()` method checks circumvent Python's normal **duck typing**. We're unable to provide instances without strictly following the class hierarchy defined by `isinstance()` checks.

We write a casino game simulation in order to experiment with endless variations on `GameStrategy`. These class definitions are very simple: they have four method definitions. There's little real benefit to using inheritance from a `GameStrategy` superclass. We should be allowed to define classes independently, not by referencing an overall superclass.

The initialization error-checking shown in this example would force us to create

subclasses merely to pass a runtime error check. No usable code is inherited from the abstract superclass.

One of the biggest duck typing issues surrounds numeric types. Different numeric types will work in different contexts. Attempts to validate the types of arguments may prevent a perfectly sensible numeric type from working properly. When attempting validation, we have the following two choices in Python:

- We write validation so that a relatively narrow collection of types is permitted, and someday the code will break because a new type that would have worked sensibly is prohibited.
- We eschew validation so that a broad collection of types is permitted, and someday the code will break because a type that does not work sensibly is used.

Note that both are essentially the same: the code could perhaps break someday. It will either break because a type is prevented from being used, even though it's sensible, or because a type that's not really sensible is used.

TIP *Just allow it*
Generally, it's considered better Python style to simply permit any type of data to be used. We'll return to this in `Chapter 5`*, The ABCs of Consistent Design.*

The question is this: why add runtime type checking when it will restrict potential future use cases?

If there's no good reason to restrict potential future use cases, then runtime type checking should be avoided.

Rather than preventing a sensible, but possibly unforeseen, use case, we provide type hints and use `mypy` to evaluate the hints. Additionally, of course, unit testing, debug logging, and documentation can help us to understand any restrictions on the types that can be processed.

With a few exceptions, all of the examples in this book use type hints to show the types of values expected and produced. The `mypy` utility can be run to confirm that the definitions are used properly. While the standard library has extensive type hints, not all packages are fully covered by hints. In `Chapter 12`, *Storing and Retrieving Objects via SQLite*, we'll use the SQLAlchemy package, which

doesn't provide complete type hints.

Initialization, encapsulation, and privacy

The general Python policy regarding privacy can be summed up as follows: *we're all adults here*.

Object-oriented design makes an explicit distinction between interface and implementation. This is a consequence of the idea of encapsulation. A class encapsulates a data structure, an algorithm, an external interface, or something meaningful. The idea is to have the capsule separate the class-based interface from the implementation details.

However, no programming language reflects every design nuance. Python doesn't typically implement all design considerations as explicit code.

One aspect of class design that is not fully carried into code is the distinction between the *private* (implementation) and *public* (interface) methods or attributes of an object. The notion of privacy in languages that support these attributes or methods (C++ or Java are two examples) is already quite complex.

These languages include settings such as private, protected, and public, as well as not specified, which is a kind of semi-private. The `private` keyword is often used incorrectly, making subclass definition needlessly difficult.

Python's notion of privacy is simple:

- Attributes and methods are all *essentially* public. The source code is available. We're all adults. Nothing can be truly hidden.
- Conventionally, we treat some names in a way that's less public. They're generally implementation details that are subject to change without notice, but there's no formal notion of private.

Names that begin with `_` are honored as being less public by some parts of Python. The `help()` function generally ignores these methods. Tools such as Sphinx can conceal these names from documentation.

Python's internal names begin (and end) with double underscores __, often pronounced *dunder*. We might call `__init__()` *dunder init*. The use of __ names is how Python internals are kept from colliding with other application features. The collection of these internal names is fully defined by the language reference. Further, there's no benefit to trying to use __ to attempt to create a *truly private* attribute or method in our code. All that happens is that we create a potential future problem if a release of Python ever starts using a name we chose for internal purposes. Also, we're likely to run afoul of the internal name mangling that is applied to these names.

The rules for the visibility of Python names are as follows:

- Most names are public.
- Names that start with _ are somewhat less public. Use them for implementation details that are truly subject to change.
- Names that begin and end with __ are internal to Python. We never make these up; we only use the names defined by the language reference.

Generally, the Python approach is to register the intent of a method (or attribute) using documentation and a well-chosen name. Often, interface methods will have elaborate documentation, possibly including `doctest` examples, while implementation methods will have more abbreviated documentation and may not have `doctest` examples.

For programmers new to Python, it's sometimes surprising that privacy is not more widely implemented. For programmers experienced in Python, it's surprising how many brain calories get burned sorting out private and public declarations that aren't really very helpful because the intent is obvious from the method names and the documentation.

Summary

In this chapter, we have reviewed the various design alternatives of the `__init__()` method. The `__init__()` method is how objects are created, and it sets the initial state of an object.

We've looked at how all Python objects are subclasses of a common parent, the `object` class, and how the default `__init__()` method for the `object` class works. This consideration leads to two design strategies for placement of the `__init__()` method:

- We can define a common `__init__()` method for all subclasses of a hierarchy. This can lead to using a factory function, separate from the `__init__()` method, to help initialize objects correctly.
- We can push the `__init__()` method into each individual subclass of a complex hierarchy, and how this changes the design of classes.

After looking at building individual objects, we looked at how we can create composite objects. This involves a number of related uses of the `__init__()` methods of different classes. More advanced topics included defining stateless objects that don't need a sophisticated initialization, using class-level (or *static*) initialization, and how to validate values before creating an invalid object.

In the next chapter, we will take a look at special methods, along with a few advanced ones as well.

Integrating Seamlessly - Basic Special Methods

There are a number of special methods that permit close integration between our classes and classes builtin Python. The Python standard library calls them **basic**. A better term might be **foundational** or **essential**. These special methods form a foundation for building classes that seamlessly integrate with other Python features.

For example, we often need string representations of a given object's value. The base class, `object`, has default implementations of `__repr__()` and `__str__()` that provide string representations of an object. Sadly, these default representations are remarkably uninformative. We'll almost always want to override one or both of these default definitions. We'll also look at `__format__()`, which is a bit more sophisticated, but serves a similar purpose.

We'll also look at other conversions, specifically `__hash__()`, `__bool__()`, and `__bytes__()`. These methods convert an object into a number, a true/false value, or a string of bytes. When we implement `__bool__()`, for example, we can use an object, `x`, in an `if` statement, as follows:

```
if x:
```

There's an implicit use of the `bool()` function, which relies on the object's implementation of the `__bool__()` special method.

We can then look at the special methods that implement the `__lt__()`, `__le__()`, `__eq__()`, `__ne__()`, `__gt__()`, and `__ge__()` comparison operators.

These basic special methods are almost always needed in class definitions.

We'll look at `__new__()` and `__del__()` last, because the use cases for these methods are rather complex. We don't need these as often as we need the other basic special methods.

We'll look in detail at how we can expand a simple class definition to add these

special methods. We'll need to look at both of the default behaviors inherited from `object` so that we can understand what overrides are needed and when they're needed.

In this chapter, we will cover the following topics:

- The `__repr__()` and `__str__()` methods
- The `__format__()` method
- The `__hash__()` method
- The `__bool__()` method
- The `__bytes__()` method
- The `comparison` operator methods
- The `__del__()` method
- The `__new__()` method and immutable objects
- The `__new__()` method and metaclasses

Technical requirements

The code files for this chapter can be found at `https://git.io/fj2UE`.

The __repr__() and __str__() methods

Python generally offers two string representations of an object. These are closely aligned with the built-in `repr()`, `str()`, and `print()` functions and the `string.format()` method:

- Generally, the `str()` method representation of an object is expected to be more friendly to humans. It is built by an object's `__str__()` method.
- The `repr()` method representation is often more technical, and typically uses a complete Python expression to rebuild an object. The documentation for the `__repr__()` method in the Python documentation (`https://docs.python.org/3/reference/datamodel.html?highlight=__del__#object.__repr__`) states the following:

  "If at all possible, this should look like a valid Python expression that could be used to recreate an object with the same value (given an appropriate environment)."

- This is built by an object's `__repr__()` method.
- The `print()` function generally uses `str()` to prepare an object for printing.
- The `format()` method of a string can also access these methods. When we use a string format line, `{x:d}`, we're providing a `"d"` parameter to the `__format__()` method of the `x` object. When we use `{x!r}` or `{x!s}` formatting, we're requesting `__repr__()` or `__str__()`, respectively.

Let's look at the default implementations first. The following is a simple class hierarchy:

```
class Card:

    def __init__(self, rank: str, suit: str) -> None:
        self.suit = suit
        self.rank = rank
        self.hard, self.soft = self._points()

    def _points(self) -> Tuple[int, int]:
        return int(self.rank), int(self.rank)

class AceCard(Card):

    def _points(self) -> Tuple[int, int]:
        return 1, 11
```

```
class FaceCard(Card):

    def _points(self) -> Tuple[int, int]:
        return 10, 10
```

We've defined three classes with four attributes in each class.

The following is an interaction with an object of one of these classes:

```
>>> x = Card('2','♠')
>>> str(x)
'<__main__.Card object at 0x1078d4518>'
>>> repr(x)
'<__main__.Card object at 0x1078d4518>'
>>> print(x)
<__main__.Card object at 0x1078d4518>
```

We can see from this output that the default implementations of `__str__()` and `__repr__()` are not very informative.

There are two broad design cases that we consider when overriding `__str__()` and `__repr__()`:

- **Simple objects**: A simple object doesn't contain a collection of other objects and generally doesn't involve very complex formatting.
- **Collection objects**: An object that contains a collection involves somewhat more complex formatting.

Simple __str__() and __repr__()

As we saw previously, the output from `__str__()` and `__repr__()` is not very informative. We'll almost always need to override them. The following is an approach to override `__str__()` and `__repr__()` when there's no collection involved. These methods belong to the `Card` class, defined previously as follows:

```
def __repr__(self) -> str:
    return f"{self.__class__.__name__}(suit={self.suit!r}, rank={self.rank!r})"

def __str__(self) -> str:
    return f"{self.rank}{self.suit}"
```

These two methods rely on `f`-strings to interpolate values from the instance into a string template. In the `__repr__()` method, the class name, suit, and rank were used to create a string that could be used to rebuild the object. In the `__str__()` method, the rank and suit were displayed in an easy-to-read form.

The template string uses two kinds of format specifications:

- The `{self.__class__.__name__}` format specification could also be written as `{self.__class__.__name__!s}` to include an explicit `!s` format specification. This is the default format, and implies using `str()` to get a string representation of the object.
- The `{self.suit!r}` and `{self.rank!r}` specifications both use the `!r` format to use the `repr()` function to get representations of the attribute values.

Collection __str__() and __repr__()

When there's a collection involved, we need to format each individual item in the collection, as well as the overall container for those items. The following is a simple collection with both the `__str__()` and `__repr__()` methods:

```
class Hand:

    def __init__(self, dealer_card: Card, *cards: Card) -> None:
        self.dealer_card = dealer_card
        self.cards = list(cards)

    def __str__(self) -> str:
        return ", ".join(map(str, self.cards))

    def __repr__(self) -> str:
        cards_text = ', '.join(map(repr, self.cards))
        return f"{self.__class__.__name__}({self.dealer_card!r}, {cards_text})"
```

The `__str__()` method has a typical recipe for applying `str()` to the items in the collection, as follows:

1. Map `str()` to each item in the collection. This will create an iterator over the resulting string values.
2. Use `", ".join()` to merge all the item strings into a single, long string.

The `__repr__()` method is a similar recipe to apply `repr()` to the items in the collection, as follows:

1. Map `repr()` to each item in the collection. This will create an iterator over the resulting string values.
2. Use `", ".join()` to merge all the item strings.
3. Use `f"{self.__class__.__name__}({self.dealer_card!r}, {cards_text})"` to combine the class name, the dealer card, and the long string of item values. This format uses `!r` formatting to ensure that the `dealer_card` attribute uses the `repr()` conversion too.

It's essential for `__str__()` to use `str()` and for `__repr__()` to use `repr()` when building representations of complex objects. This simple consistency guarantees that results from very complex objects with multiple layers of nesting will have consistent string values.

The __format__() method

The `__format__()` method is used by `f`-strings, the `str.format()` method, as well as the `format()` built-in function. All three of these interfaces are used to create presentable string versions of a given object.

The following are the two ways in which arguments will be presented to the `__format__()` method of a `someobject` object:

- `someobject.__format__("")`: This happens when an application uses a string such as `f"{someobject}"`, a function such as `format(someobject)`, or a string formatting method such as `"{0}".format(someobject)`. In these cases, there was no `:` in the format specification, so a default zero-length string is provided as the argument value. This should produce a default format.
- `someobject.__format__(spec)`: This happens when an application uses a string such as `f"{someobject:spec}"`, a function such as `format(someobject, spec)`, or something equivalent to the `"{0:spec}".format(someobject)` string method.

Note that an `f`-string `f"{item!r}"` with `!r` or a `"{0!r}".format(item)` format method with `!r` doesn't use the object's `__format__()` method. The portion after `!` is a conversion specification.

A conversion specification of `!r` uses the `repr()` function, which is generally implemented by an object's `__repr__()` method. Similarly, a conversion specification of `!s` uses the `str()` function, which is implemented by the `__str__()` method. The `!a` conversion specification uses the `ascii()` function. The `ascii()` function generally depends on the `__repr__()` method to provide the underlying representation.

With a specification of `""`, a sensible implementation is `return str(self)`. This provides an obvious consistency between the various string representations of an object.

The format specification provided as the argument value to `__format__()` will be all the text after `":"` in the original format string. When we write `f"{value:06.4f}"`, `06.4f` is the format specification that applies to the `item` value in the string to be

formatted.

Section 2.4.3 of the *Python Language Reference* defines the formatted string (f-string) mini-language. Each format specification has the following syntax:

```
[[fill]align][sign][#][0][width][grouping_option][.precision][type]
```

We can parse these potentially complex standard specifications with a **regular expression** (**RE**), as shown in the following code snippet:

```
re.compile(
    r"(?P<fill_align>.?[\<\>=\^])?"
    r"(?P<sign>[-+ ])?"
    r"(?P<alt>#)?"
    r"(?P<padding>0)?"
    r"(?P<width>\d*)"
    r"(?P<grouping_option>,)?"
    r"(?P<precision>\.\d*)?"
    r"(?P<type>[bcdeEfFgGnosxX%])?")
```

This RE will break the specification into eight groups. The first group will have both the `fill` and `alignment` fields from the original specification. We can use these groups to work out the formatting for the attributes of classes we've defined.

In some cases, this very general format specification mini-language might not apply to the classes that we've defined. This leads to a need to define a format specification mini-language and process it in the customized `__format__()` method.

As an example, here's a trivial language that uses the `%r` character to show us the rank and the `%s` character to show us the suit of an instance of the `Card` class. The `%%` character becomes `%` in the resulting string. All other characters are repeated literally.

We could extend our `Card` class with formatting, as shown in the following code snippet:

```
def __format__(self, format_spec: str) -> str:
    if format_spec == "":
        return str(self)
    rs = (
        format_spec.replace("%r", self.rank)
                   .replace("%s", self.suit)
                   .replace("%%", "%")
    )
    return rs
```

This definition checks for a format specification. If there's no specification, then

the `str()` function is used. If a specification was provided, a series of replacements is done to fold rank, suit, and any `%` characters into the format specification, turning it into the output string.

This allows us to format cards as follows:

```
print( "Dealer Has {0:%r of %s}".format( hand.dealer_card) )
```

The format specification (`"%r of %s"`) is passed to our `__format__()` method as the `format` parameter. Using this, we're able to provide a consistent interface for the presentation of the objects of the classes that we've defined.

Alternatively, we can define things as follows:

```
    default_format = "some specification"
    def __str__(self) -> str:
        return self.__format__(self.default_format)
    def __format__(self, format_spec: str) -> str:
        if format_spec == "":
            format_spec = self.default_format
       # process using format_spec.
```

This has the advantage of putting all string presentations into the `__format__()` method instead of spreading them between `__format__()` and `__str__()`. This does have a disadvantage, because we don't always need to implement `__format__()`, but we almost always need to implement `__str__()`.

Nested formatting specifications

The `string.format()` method can handle nested instances of `{}` to perform simple keyword substitution into the format specification itself. This replacement is done to create the final format string that's passed to our class's `__format__()` method. This kind of nested substitution simplifies some kinds of relatively complex numeric formatting by parameterizing an otherwise generic specification.

The following is an example where we've made `width` easy to change in the `format` parameter:

```
width = 6
for hand, count in statistics.items():
    print(f"{hand} {count:{width}d}")
```

We've defined a generic format, `f"{hand} {count:{width}d}"`, which requires a `width` parameter to make it into a final format specification. In the example, `width` is `6`, which means that the final format will be `f"{hand} {count:6d}"`. The expanded format string, `"6d"` will be the specification provided to the `__format__()` method of the underlying object.

Collections and delegating format specifications

When formatting a complex object that includes a collection, we have two formatting issues: how to format the overall object and how to format the items in the collection. When we look at `Hand`, for example, we see that we have a collection of individual `Cards` objects. We'd like to have `Hand` delegate some formatting details to the individual `Card` instances in the `Hand` collection.

The following is a `__format__()` method that applies to `Hand`:

```
def __format__(self, spec: str) -> str:
    if spec == "":
        return str(self)
    return ", ".join(f"{c:{spec}}" for c in self.cards)
```

The `spec` parameter will be used for each individual `Card` instance within the `Hand` collection. `f`-string `f"{c:{spec}}"` uses the nested format specification technique to push the `spec` string into the format. This creates a final format, which will be applied to each `Card` instance.

Given this method, we can format a `Hand` object, `player_hand`, as follows:

```
>>> d = Deck()
>>> h = Hand(d.pop(), d.pop(), d.pop())
>>> print("Player: {hand:%r%s}".format(hand=h))
Player: K♦, 9♥
```

This string in the `print()` function used the `format()` method of the `Hand` object. This passed the `%r%s` format specification to each `Card` instance of the `Hand` object to provide the desired formatting for each card of the hand.

The __hash__() method

The built-in `hash()` function invokes the `__hash__()` method of a given object. This hash is a calculation that reduces a (potentially complex) value to a small integer value. Ideally, a hash reflects all the bits of the source value. Other hash calculations – often used for cryptographic purposes – can produce very large values.

Python includes two `hash` libraries. The cryptographic-quality hash functions are in `hashlib`. The `zlib` module also has two high-speed hash functions: `adler32()` and `crc32()`. For the most common cases, we don't use any of these library functions. They're only needed to hash extremely large, complex objects.

The `hash()` function (and the associated `__hash__()` method) is used to create a small integer key that is used to work with collections, such as `set`, `frozenset`, and `dict`. These collections use the `hash` value of an **immutable** object to rapidly locate the object in the collection.

Immutability is important here; we'll mention it many times. Immutable objects don't change their state. The number `3`, for example, cannot meaningfully change state. It's always `3`. Similarly, more complex objects can have an immutable state. Python strings are immutable. These can then be used as keys to mappings and sets.

The default `__hash__()` implementation inherited from an object returns a value based on the object's internal ID value. This value can be seen with the `id()` function, as follows:

```
>>> x = object()
>>> hash(x)
280696151
>>> id(x)
4491138416
>>> id(x) / 16
280696151.0
```

The `id` values shown in this example will vary from system to system.

From this, we can see that, on the author's system, the hash value is the object's

`id//16`. This detail might vary from platform to platform.

What's essential is the strong correlation between the internal ID and the default `__hash__()` method. This means that the default behavior is for each object to be hashable as well as utterly distinct, even if the objects appear to have the same value.

We'll need to modify this if we want to coalesce different objects with the same value into a single hashable object. We'll look at an example in the next section, where we would like two instances of a single `Card` instance to be treated as if they were the same object.

Deciding what to hash

Not every object should provide a hash value. Specifically, if we're creating a class of stateful, **mutable** objects, the class should *never* return a hash value. There should not be an implementation of the `__hash__()` method.

Immutable objects, on the other hand, might sensibly return a hash value so that the object can be used as the key in a dictionary, or as a member of a set. In this case, the hash value needs to parallel the way the test for equality works. It's bad to have objects that claim to be equal and have different hash values. The reverse —objects with the same hash that are actually not equal – is acceptable and expected. Several distinct objects may happen to have overlapping hash values.

There are three tiers of equality comparison:

- **The same hash value**: This means that two objects *could* be equal. The hash value provides us with a quick check for likely equality. If the hash value is different, the two objects cannot possibly be equal, nor can they be the same object.
- **Compare as equal**: This means that the hash values must also have been equal. This is the definition of the `==` operator. The objects may be the same object.
- **Same ID value**: This means that they are the same object. They also compare as equal and will also have the same hash value. This is the definition of the `is` operator.

The **Fundamental Law of Hash** (**FLH**) has two parts:

- Objects that compare as equal have the same hash value.
- Objects with the same hash value may actually be distinct and not compare as equal.

We can think of a hash comparison as being the first step in an equality test. This is a very fast comparison to determine whether subsequent equality checks are necessary.

The `__eq__()` method, which we'll also look at in the section on comparison operators, is intimately tied up with hashing. This provides a potentially slow field-by-field comparison between objects.

Here's a contrived example of two distinct numeric values with the same hash value:

```
>>> v1 = 123_456_789
>>> v2 = 2_305_843_009_337_150_740
>>> hash(v1)
123456789
>>> hash(v2)
123456789
>>> v2 == v1
False
```

Notice that a `v1` integer, equal to `123,456,789`, has a hash value equal to itself. This is typical of integers up to the hash modulus. The `v2` integer has the same hash, but a different actual value.

This hash collision is expected. It's part of the known processing overhead when creating sets or dictionaries. There will be unequal objects that are reduced to coincidentally equal hash values.

There are three use cases for defining equality tests and hash values via the `__eq__()` and `__hash__()` method functions:

- **Immutable objects**: These are stateless objects of types such as tuples, namedtuples, and frozensets that cannot be updated. We have two choices:
    - Define neither `__hash__()` nor `__eq__()`. This means doing nothing and using the inherited definitions. In this case, `__hash__()` returns a trivial function of the ID value for the object, and `__eq__()` compares the ID values.
    - Define both `__hash__()` and `__eq__()`. Note that we're expected to define both for an immutable object.
- **Mutable objects**: These are stateful objects that can be modified internally. We have one design choice:
    - Define `__eq__()` but set `__hash__` to `None`. These cannot be used as `dict` keys or items in `sets`.

The default behavior for immutable objects will be undesirable when an application requires two distinct objects to compare as equal. For example we

might want two instances of `Card(1, Clubs)` to test as equal and compute the same hash; this will not happen by default. For this to work, we'll need to override the `__hash__()` and `__eq__()` methods.

Note that there's an additional possible combination: defining `__hash__()` but using a default definition for `__eq__()`. This is simply a waste of code, as the default `__eq__()` method is the same as the `is` operator. Using the default `__hash__()` method would have involved writing less code for the same behavior.

We'll look at each of these three situations in detail.

Inheriting definitions for immutable objects

Let's see how default definitions operate. The following is a simple `class` hierarchy that uses the default definitions of `__hash__()` and `__eq__()`:

```
class Card:
    insure = False

    def __init__(self, rank: str, suit: "Suit", hard: int, soft: int) -> None:
        self.rank = rank
        self.suit = suit
        self.hard = hard
        self.soft = soft

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}(suit={self.suit!r}, rank={self.rank!r})"

    def __str__(self) -> str:
        return f"{self.rank}{self.suit}"


class NumberCard(Card):

    def __init__(self, rank: int, suit: "Suit") -> None:
        super().__init__(str(rank), suit, rank, rank)


class AceCard(Card):
    insure = True

    def __init__(self, rank: int, suit: "Suit") -> None:
        super().__init__("A", suit, 1, 11)


class FaceCard(Card):

    def __init__(self, rank: int, suit: "Suit") -> None:
        rank_str = {11: "J", 12: "Q", 13: "K"}[rank]
        super().__init__(rank_str, suit, 10, 10)
```

This is a `class` hierarchy for *philosophically* immutable objects. We haven't taken care to implement the special methods that prevent the attributes from getting updated. We'll look at attribute access in the next chapter. Here's the definition of the enumerated class of `Suit` values:

```
from enum import Enum
class Suit(str, Enum):
    Club = "\N{BLACK CLUB SUIT}"
    Diamond = "\N{BLACK DIAMOND SUIT}"
    Heart = "\N{BLACK HEART SUIT}"
```

```
    Spade = "\N{BLACK SPADE SUIT}"
```

Let's see what happens when we use this `class` hierarchy:

```
>>> c1 = AceCard(1, Suit.Club)
>>> c2 = AceCard(1, Suit.Club)
```

We defined two instances of what appear to be the same `Card` instance. We can check the `id()` values as shown in the following code snippet:

```
>>> id(c1), id(c2)
(4492886928, 4492887208)
```

They have different `id()` numbers; this means that they're distinct objects. This meets our expectations.

We can check to see whether they're the same by using the `is` operator as shown in the following code snippet:

```
>>> c1 is c2
False
```

The `is` test is based on the `id()` numbers; it shows us that they are, indeed, separate objects.

We can see that their hash values are different from each other:

```
>>> hash(c1), hash(c2)
(-9223372036575077572, 279698247)
```

These hash values come directly from the `id()` values. This was our expectation for the inherited methods. In this implementation, we can compute the hash from the `id()` function, as shown in the following code snippet:

```
>>> id(c1) / 16
268911077.0
>>> id(c2) / 16
268911061.0
```

As the hash values are different, they must not compare as equal. While this fits the definitions of hash and equality, it violates our expectations for instances of this class.

We created the two objects with the same attributes. The following is an equality check:

```
>>> c1 == c2
False
```

Even though the objects have the same attribute values, they don't compare as equal. In some applications, this might not be good. For example, when accumulating statistical counts of dealer cards, we don't want to have six counts for one card because the simulation used a six-deck shoe.

We can see that these are proper immutable objects. The following example shows that these objects can be put into a set:

```
>>> set([c1, c2])
{AceCard(suit=<Suit.Club: '♣'>, rank='A'), AceCard(suit=<Suit.Club: '♣'>, rank='A')}
```

This is the documented behavior from the *Standard Library Reference* documentation. By default, we'll get a `__hash__()` method based on the ID of the object so that each instance appears unique. However, this isn't always what we want.

Overriding definitions for immutable objects

The following is a simple `class` hierarchy that provides us with definitions of `__hash__()` and `__eq__()`:

```
import sys
class Card2:
    insure = False

    def __init__(self, rank: str, suit: "Suit", hard: int, soft: int) -> None:
        self.rank = rank
        self.suit = suit
        self.hard = hard
        self.soft = soft

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}(suit={self.suit!r}, rank={self.rank!r})"

    def __str__(self) -> str:
        return f"{self.rank}{self.suit}"

    def __eq__(self, other: Any) -> bool:
        return (
            self.suit == cast(Card2, other).suit
            and self.rank == cast(Card2, other).rank
        )

    def __hash__(self) -> int:
        return (hash(self.suit) + 4*hash(self.rank)) % sys.hash_info.modulus

class AceCard2(Card2):
    insure = True

    def __init__(self, rank: int, suit: "Suit") -> None:
        super().__init__("A", suit, 1, 11)
```

This object is immutable in principle. There's no formal mechanism to the immutable instances. We'll look at how to prevent attribute value changes in `Chapter 4`, *Attribute Access, Properties, and Descriptors*.

Also, note that the preceding code omits two of the subclasses that didn't change significantly from the previous example, `FaceCard` and `NumberCard`.

The `__eq__()` method has a type hint, which suggests that it will compare an object of any class and return a `bool` result. The implementation uses a `cast()` function to provide a hint to **mypy** that the value of `other` will always be an instance of `Card2`

or a runtime type error can be raised. The `cast()` function is part of mypy's type hinting and has no runtime effect of any kind. The function compares two essential values: `suit` and `rank`. It doesn't need to compare the hard and soft values; they're derived from `rank`.

The rules for *Blackjack* make this definition a bit suspicious. Suit doesn't actually matter in *Blackjack*. Should we merely compare rank? Should we define an additional method that compares rank only? Or, should we rely on the application to compare rank properly? There's no best answer to these questions; these are potential design trade-offs.

The `__hash__()` function computes a unique value pattern from the two essential attributes. This computation is based on the hash values for the rank and the suit. The rank will occupy the most significant bits of the value, and suit will be the least significant bits. This tends to parallel the way that cards are ordered, with rank being more important than suit. The hash values must be computed using the `sys.hash_info.modulus` value as a modulus constraint.

Let's see how objects of these classes behave. We expect them to compare as equal and behave properly with sets and dictionaries. Here are two objects:

```
>>> c1 = AceCard2(1, '♣')
>>> c2 = AceCard2(1, '♣')
```

We defined two instances of what appear to be the same card. We can check the ID values to be sure that they're distinct objects:

```
>>> id(c1), id(c2)
(4302577040, 4302577296)
>>> c1 is c2
False
```

These have different `id()` numbers. When we test with the `is` operator, we see that they're distinct objects. This fits our expectations so far.

Let's compare the hash values:

```
>>> hash(c1), hash(c2)
(1259258073890, 1259258073890)
```

The hash values are identical. This means that they could be equal.

The equality operator shows us that they properly compare as equal:

```
>>> c1 == c2
True
```

Because the objects produce a hash value, we can put them into a set, as follows:

```
>>> set([c1, c2])
{AceCard2(suit=<Suit.Club: '♣'>, rank='A')}
```

Since the two objects create the same hash value and test as equal, they appear to be two references to the same object. Only one of them is kept in the set. This meets our expectations for complex immutable objects. We had to override both special methods to get consistent, meaningful results.

Overriding definitions for mutable objects

This example will continue using the `Cards` class. The idea of mutable cards is strange, perhaps even wrong. However, we'd like to apply just one small tweak to the previous examples.

The following is a class hierarchy that provides us with the definitions of `__hash__()` and `__eq__()`, appropriate for mutable objects. The parent class is as follows:

```
class Card3:
    insure = False

    def __init__(self, rank: str, suit: "Suit", hard: int, soft: int) -> None:
        self.rank = rank
        self.suit = suit
        self.hard = hard
        self.soft = soft

    def __repr__(self) -> str:
        return f"{self.__class__.__name__}(suit={self.suit!r}, rank={self.rank!r})"

    def __str__(self) -> str:
        return f"{self.rank}{self.suit}"

    def __eq__(self, other: Any) -> bool:
        return (
            self.suit == cast(Card3, other).suit
            and self.rank == cast(Card3, other).rank
        )
```

A subclass of `Card3` is shown in the following example:

```
class AceCard3(Card3):
    insure = True

    def __init__(self, rank: int, suit: "Suit") -> None:
        super().__init__("A", suit, 1, 11)
```

Let's see how objects of these classes behave. We expect them to compare as equal but not work at all with sets or dictionaries. We'll create two objects as follows:

```
>>> c1 = AceCard3(1, '♣')
>>> c2 = AceCard3(1, '♣')
```

We've defined two instances of what appear to be the same card.

We'll look at their ID values to ensure that they really are distinct:

```
>>> id(c1), id(c2)
(4302577040, 4302577296)
```

No surprise here. Now, we'll see if we can get hash values:

```
>>> hash(c1), hash(c2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'AceCard3'
```

The `Card3` objects can't be hashed. They won't provide a value for the `hash()` function. This is the expected behavior. However, we can perform equality comparisons, as shown in the following code snippet:

```
>>> c1 == c2
True
```

The equality test works properly, allowing us to compare cards. They just can't be inserted into sets or used as keys in a dictionary.

The following is what happens when we try to put them into a set:

```
>>> set([c1, c2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'AceCard3'
```

We get a proper exception when trying to do so.

Clearly, this is not a proper definition for something that – in real life – is immutable, such as a card. This style of definition is more appropriate for stateful objects, such as `Hand`, where the content of the hand is always changing. We'll provide you with a second example of stateful objects in the following section.

Making a frozen hand from a mutable hand

If we want to perform a statistical analysis of specific `Hand` instances, we might want to create a dictionary that maps a `Hand` instance to a count. We can't use a mutable `Hand` class as the key in a mapping. We can, however, parallel the design of `set` and `frozenset` and create two classes: `Hand` and `FrozenHand`. This allows us to *freeze* a `Hand` instance by creating `FrozenHand`; the frozen version is immutable and can be used as a key in a dictionary.

The following is a simple `Hand` definition:

```
class Hand:

    def __init__(self, dealer_card: Card2, *cards: Card2) -> None:
        self.dealer_card = dealer_card
        self.cards = list(cards)

    def __str__(self) -> str:
        return ", ".join(map(str, self.cards))

    def __repr__(self) -> str:
        cards_text = ", ".join(map(repr, self.cards))
        return f"{self.__class__.__name__}({self.dealer_card!r}, {cards_text})"

    def __format__(self, spec: str) -> str:
        if spec == "":
            return str(self)
        return ", ".join(f"{c:{spec}}" for c in self.cards)

    def __eq__(self, other: Any) -> bool:
        if isinstance(other, int):
            return self.total() == cast(int, other)
        try:
            return (
                self.cards == cast(Hand, other).cards
                and self.dealer_card == cast(Hand, other).dealer_card
            )
        except AttributeError:
            return NotImplemented
```

This is a mutable object; it does not compute a hash value, and can't be used in a set or dictionary key. It does have a proper equality test that compares two hands. As with previous examples, the parameter to the `__eq__()` method has a type hint of `Any`, and a do-nothing `cast()` function is used to tell the mypy program that the argument values will always be instances of `Hand`. The following is a

frozen version of `Hand`:

```
import sys

class FrozenHand(Hand):

    def __init__(self, *args, **kw) -> None:
        if len(args) == 1 and isinstance(args[0], Hand):
            # Clone a hand
            other = cast(Hand, args[0])
            self.dealer_card = other.dealer_card
            self.cards = other.cards
        else:
            # Build a fresh Hand from Card instances.
            super().__init__(*args, **kw)

    def __hash__(self) -> int:
        return sum(hash(c) for c in self.cards) % sys.hash_info.modulus
```

The frozen version has a constructor that will build one `Hand` class from another `Hand` class. It defines a `__hash__()` method that sums the card's hash value, which is limited to the `sys.hash_info.modulus` value. For the most part, this kind of modulus-based calculation works out well for computing the hashes of composite objects. We can now use these classes for operations such as the following code snippet:

```
from collections import defaultdict
stats = defaultdict(int)

d = Deck()
h = Hand(d.pop(), d.pop(), d.pop())
h_f = FrozenHand(h)
stats[h_f] += 1
```

We've initialized a statistics dictionary, `stats`, as a `defaultdict` dictionary that can collect integer counts. We could also use a `collections.Counter` object for this.

By freezing an instance of the `Hand` class, we can compute a hash and use it as a key in a dictionary. This makes it easy to create a `defaultdict` for collecting counts of each hand that actually gets dealt.

The __bool__() method

Python has a pleasant definition of falsity. The reference manual lists a large number of values that will test as equivalent to `False`. This includes things such as `False`, `0`, `''`, `()`, `[]`, and `{}`. Objects not included in this list will test as equivalent to `True`.

Often, we'll want to check for an object being *not empty* with a simple statement, as follows:

```
if some_object:
    process(some_object)
```

Under the hood, this is the job of the `bool()` built-in function. This function depends on the `__bool__()` method of a given object.

The default `__bool__()` method returns as `True`. We can see this with the following code:

```
>>> x = object()
>>> bool(x)
True
```

For most classes, this is perfectly valid. Most objects are not expected to be `False`. For collections, however, the default behavior is not appropriate. An empty collection should be equivalent to `False`. A non-empty collection should return `True`. We might want to add a method like this to our `Deck` objects.

If the implementation of a collection involves wrapping a list, we might have something as shown in the following code snippet:

```
def __bool__(self):
    return bool(self._cards)
```

This delegates the Boolean function to the internal `self._cards` collection.

If we're extending a list, we might have something as follows:

```
def __bool__(self):
    return super().__bool__(self)
```

This delegates to the superclass definition of the `__bool__()` function.

In both cases, we're specifically delegating the Boolean test. In the wrap case, we're delegating to the collection. In the extend case, we're delegating to the superclass. Either way, wrap or extend, an empty collection will be `False`. This will give us a way to see whether the `Deck` object has been entirely dealt and is empty.

We can do this as shown in the following code snippet:

```
d = Deck()
while d:
    card = d.pop()
    # process the card
```

This loop will deal all the cards without getting an `IndexError` exception when the deck has been exhausted.

The __bytes__() method

There are relatively few occasions when you will need to transform an object into bytes. Bytes representation is used for the serialization of objects for persistent storage or transfer. We'll look at this in detail in `Chapter 10`, *Serializing and Saving - JSON, YAML, Pickle, CSV and XML* through `Chapter 14`, *Configuration Files and Persistence*.

In the most common situation, an application will create a string representation, and the built-in encoding capabilities of the Python IO classes can be used to transform the string into bytes. This works perfectly for almost all situations. The main exception would be when we're defining a new kind of string. In which case, we'd need to define the encoding of that string.

The `bytes()` function does a variety of things, depending on the arguments:

- `bytes(integer)`: This returns an immutable bytes object with the given number of `0x00` values.
- `bytes(string)`: This will encode the given string into bytes. Additional parameters for encoding and error handling will define the details of the encoding process.
- `bytes(something)`: This will invoke `something.__bytes__()` to create a bytes object. The encoding of error arguments will not be used here.

The base `object` class does not define `__bytes__()`. This means our classes don't provide a `__bytes__()` method by default.

There are some exceptional cases where we might have an object that will need to be encoded directly into bytes before being written to a file. It's often simpler to work with strings and allow the `str` type to produce bytes for us. When working with bytes, it's important to note that there's no simple way to decode bytes from a file or interface. The built-in `bytes` class will only decode strings, not our unique, new objects. This means that we'll need to parse the strings that are decoded from the bytes. Or, we might need to explicitly parse the bytes using the `struct` module and create our unique objects from the parsed values.

We'll look at encoding and decoding the `Card2` instance into bytes. As there are only 52 card values, each card could be packed into a single byte. However, we've elected to use a character to represent `suit` and a character to represent `rank`. Further, we'll need to properly reconstruct the subclass of `Card2`, so we have to encode several things:

- The subclass of `Card2` (`AceCard2`, `NumberCard2`, and `FaceCard2`)
- The parameters to the subclass-defined `__init__()` methods.

Note that some of our alternative `__init__()` methods will transform a numeric rank into a string, losing the original numeric value. For the purposes of reversible byte encoding, we need to reconstruct this original numeric rank value.

The following is an implementation of `__bytes__()`, which returns a `utf-8` encoding of the `Card2` subclass name, `rank`, and `suit`:

```
def __bytes__(self) -> bytes:
    class_code = self.__class__.__name__[0]
    rank_number_str = {"A": "1", "J": "11", "Q": "12", "K": "13"}.get(
        self.rank, self.rank
    )
    string = f"({' '.join([class_code, rank_number_str, self.suit])})"
    return bytes(string, encoding="utf-8")
```

This works by creating a string representation of the `Card2` object. The representation uses the `()` objects to surround three space-separated values: code that represents the class, a string that represents the rank, and the suit. This string is then encoded into bytes.

The following snippet shows how bytes representation looks:

```
>>> c1 = AceCard2(1, Suit.Club)
>>> bytes(c1)
b'(A 1 \xe2\x99\xa3)'
```

When we are given a pile of bytes, we can decode the string from the bytes and then parse the string into a new `Card2` object. The following is a method that can be used to create a `Card2` object from bytes:

```
def card_from_bytes(buffer: bytes) -> Card2:
    string = buffer.decode("utf8")
    try:
        if not (string[0] == "(" and string[-1] == ")"):
            raise ValueError
```

```
        code, rank_number, suit_value = string[1:-1].split()
        if int(rank_number) not in range(1, 14):
            raise ValueError
        class_ = {"A": AceCard2, "N": NumberCard2, "F": FaceCard2}[code]
        return class_(int(rank_number), Suit(suit_value))
    except (IndexError, KeyError, ValueError) as ex:
        raise ValueError(f"{buffer!r} isn't a Card2 instance")
```

In the preceding code, we've decoded the bytes into a string. We checked the string for required `()`. We've then parsed the string into three individual values using `string[1:-1].split()`. From those values, we converted the rank to an integer of the valid range, located the class, and built an original `Card2` object.

We can reconstruct a `Card2` object from a stream of bytes as follows:

```
>>> data = b'(N 5 \xe2\x99\xa5)'
>>> c2 = card_from_bytes(data)
>>> c2
NumberCard2(suit=<Suit.Heart: '♥'>, rank='5')
```

It's important to note that an external bytes representation is often challenging to design. In all cases, the bytes are the representation of the state of an object. Python already has a number of representations that work well for a variety of class definitions.

It's often better to use the `pickle` or `json` modules than to invent a low-level bytes representation of an object. This will be the subject of `Chapter 10`, *Serializing and Saving - JSON, YAML, Pickle, CSV, and XML*.

The comparison operator methods

Python has six comparison operators. These operators have special method implementations. According to the documentation, mapping works as follows:

- `x<y` is implemented by `x.__lt__(y)`.
- `x<=y` is implemented by `x.__le__(y)`.
- `x==y` is implemented by `x.__eq__(y)`.
- `x!=y` is implemented by `x.__ne__(y)`.
- `x>y` is implemented by `x.__gt__(y)`.
- `x>=y` is implemented by `x.__ge__(y)`.

We'll return to comparison operators again when looking at numbers in Chapter 8, *Creating Numbers*.

There's an additional rule regarding what operators are actually implemented that's relevant here. These rules are based on the idea that the object's class on the left-hand side of an operator defines the required special method. If it doesn't, Python can try an alternative operation by changing the order and considering the object on the right-hand side of the operator.

Here are the two basic rules
First, the operand on the left-hand side of the operator is checked for an implementation: `A<B` means `A.__lt__(B)`.
Second, the operand on the right-hand side of the operator is checked for a reversed implementation: `A<B` means `B.__gt__(A)`.
The rare exception to this occurs when the right-hand operand is a subclass of the left-hand operand; then, the right-hand operand is checked first to allow a subclass to override a superclass.

We can see how this works by defining a class with only one of the operators defined and then using it for other operations.

The following is a partial class that we can use:

```
class BlackJackCard_p:

    def __init__(self, rank: int, suit: Suit) -> None:
        self.rank = rank
        self.suit = suit
```

```
    def __lt__(self, other: Any) -> bool:
        print(f"Compare {self} < {other}")
        return self.rank < cast(BlackJackCard_p, other).rank

    def __str__(self) -> str:
        return f"{self.rank}{self.suit}"
```

This follows the *Blackjack* comparison rules, where suits don't matter and cards are only compared by their rank. We've omitted all but one of the comparison methods to see how Python will fall back when an operator is missing. This class will allow us to perform `<` comparisons. Interestingly, Python can also use this to perform `>` comparisons by switching the argument order. In other words, $x < y \equiv y > x$. This is the mirror reflection rule; we'll see it again in Chapter 8, *Creating Numbers*.

We see this when we try to evaluate different comparison operations. We'll create two `BlackJackCard_p` instances and compare them in various ways, as shown in the following code snippet:

```
>>> two = BlackJackCard_p(2, Suit.Spade)
>>> three = BlackJackCard_p(3, Suit.Spade)
>>> two < three
Compare 2♠ < 3♠
True
>>> two > three
Compare 3♠ < 2♠
False
>>> two == three
False
```

This example shows that a comparison using the `<` operator is implemented by the defined `__lt__()` method, as expected. When using the `>` operator, then the available `__lt__()` method is also used, but with the operands reversed.

What happens when we try to use an operator such as `<=`? This shows the exception:

```
>>> two <= three
Traceback (most recent call last):
 File "/Users/slott/miniconda3/envs/py37/lib/python3.7/doctest.py", line 1329, in __run
 compileflags, 1), test.globs)
 File "<doctest __main__.__test__.test_blackjackcard_partial[5]>", line 1, in <module>
 print("{0} <= {1} :: {2!r}".format(two, three, two <= three)) # doctest: +IGNORE_EXCEPT
TypeError: '<=' not supported between instances of 'BlackJackCard_p' and 'BlackJackCard_
```

From this, we can see where `two < three` maps to `two.__lt__(three)`.

However, for `two > three`, there's no `__gt__()` method defined; Python uses

`three.__lt__(two)` as a fallback plan.

By default, the `__eq__()` method is inherited from `object`. You will recall that the default implementation compares the object IDs and all unique objects will compare as not equal. The objects participate in `==` tests as follows:

```
>>> two_c = BlackJackCard_p(2, Suit.Club)
>>> two_c == BlackJackCard_p(2, Suit.Club)
False
```

We can see that the results aren't quite what we expect. We'll often need to override the default implementation of `__eq__()`.

There's no logical connection among the operators either. Mathematically, we can derive all the necessary comparisons from just two. Python doesn't do this automatically. Instead, Python handles the following four simple reflection pairs by default:

$$x < y \equiv y > x$$

$$x \leq y \equiv y \geq x$$

$$x = y \equiv y = x$$

$$x \neq y \equiv y \neq x$$

This means that we must, at a minimum, provide one from each of the four pairs. For example, we could provide `__eq__()`, `__ne__()`, `__lt__()`, and `__le__()`.

The `@functools.total_ordering` decorator can help overcome the default limitation. This decorator deduces the rest of the comparisons from just `__eq__()` and one of these: `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()`. This provides all the necessary comparisons. We'll revisit this in `Chapter 8`, *Creating Numbers*.

Designing comparisons

There are two considerations when defining comparison operators:

- The obvious question of how to compare two objects of the same class.
- The less obvious question of how to compare objects of different classes.

For a class with multiple attributes, we often have a profound ambiguity when looking at the comparison operators. It might not be perfectly clear which of the available attributes participate in the comparison.

Consider the humble playing card (again!). An expression, such as `card1 == card2`, is clearly intended to compare both `rank` and `suit`, right? Is that always true? After all, `suit` doesn't matter in games such as *Blackjack*.

If we want to decide whether a `Hand` object can be split, we must decide whether the split operation is valid. In *Blackjack*, a hand can only be split if the two cards are of the same rank. The implementation we chose for equality testing will then change how we implement the rules for splitting a hand.

This leads to some alternatives. In one case, the use of rank is implicit; the other requires it to be explicit. The following is the first code snippet for rank comparison:

```
if hand.cards[0] == hand.cards[1]
```

The following is the second code snippet for rank comparison:

```
if hand.cards[0].rank == hand.cards[1].rank
```

While one is shorter, brevity is not always best. If we define equality to only consider `rank`, we may have trouble creating unit tests. If we use only rank, then `assert expectedCard == actualCard` will tolerate a wide variety of cards when a unit test should be focused on exactly correct cards.

An expression such as `card1 <= 7` is clearly intended to compare `rank`. Should the ordering operators have slightly different semantics than equality testing?

There are more trade-off questions that stem from a rank-only comparison. How could we order cards by `suit` if this attribute is not used for ordering comparisons?

Furthermore, equality checks must parallel the hash calculation. If we've included multiple attributes in the hash, we also need to include them in the equality comparison. In this case, it appears that equality (and inequality) between cards must be full `Card` comparisons, because we're hashing the `Card` values to include `rank` and `suit`.

The ordering comparisons between `Card`, however, could be `rank` only. Comparisons against integers could similarly be `rank` only. For the special case of detecting a split, `hand.cards[0].rank == hand.cards[1].rank` could be used, because it states the rule for a valid split explicitly.

Implementation of a comparison of objects of the same class

We'll look at a simple same-class comparison by looking at a more complete `BlackJackCard` class:

```
class BlackJackCard:

    def __init__(self, rank: int, suit: Suit, hard: int, soft: int) -> None:
        self.rank = rank
        self.suit = suit
        self.hard = hard
        self.soft = soft

    def __lt__(self, other: Any) -> bool:
        if not isinstance(other, BlackJackCard):
            return NotImplemented
        return self.rank < other.rank

    def __le__(self, other: Any) -> bool:
        try:
            return self.rank <= cast(BlackJackCard, other).rank
        except AttributeError:
            return NotImplemented

    def __gt__(self, other: Any) -> bool:
        if not isinstance(other, BlackJackCard):
            return NotImplemented
        return self.rank > other.rank

    def __ge__(self, other: Any) -> bool:
        try:
            return self.rank >= cast(BlackJackCard, other).rank
        except AttributeError:
            return NotImplemented

    def __eq__(self, other: Any) -> bool:
        if not isinstance(other, BlackJackCard):
            return NotImplemented
        return (self.rank == other.rank
                and self.suit == other.suit)

    def __ne__(self, other: Any) -> bool:
        if not isinstance(other, BlackJackCard):
            return NotImplemented
        return (self.rank != other.rank
                or self.suit != other.suit)

    def __str__(self) -> str:
        return f"{self.rank}{self.suit}"

    def __repr__(self) -> str:
        return (f"{self.__class__.__name__}"
                f"(rank={self.rank!r}, suit={self.suit!r}, "
```

```
                f"hard={self.hard!r}, soft={self.soft!r})")
```

This example class defines all six comparison operators.

The various comparison methods use two kinds of type checking: **class** and **protocol**:

- Class-based type checking uses `isinstance()` to check the class membership of the object. When the check fails, the method returns the special `NotImplemented` value; this allows the other operand to implement the comparison. The `isinstance()` check also informs mypy of a type constraint on the objects named in the expression.
- Protocol-based type checking follows **duck typing** principles. If the object supports the proper protocol, it will have the necessary attributes. This is shown in the implementation of the `__le__()` and `__ge__()` methods. A `try:` block is used to wrap the attempt and provide a useful `NotImplemented` value if the protocol isn't available in the object. In this case, the `cast()` function is used to inform mypy that only objects with the expected class protocol will be used at runtime.

There's a tiny conceptual advantage to checking for support for a given protocol instead of membership in a class: it avoids needlessly over-constraining operations. It's entirely possible that someone might want to invent a variation on a card that follows the protocol of `BlackJackCard`, but is not defined as a proper subclass of `BlackjackCard`. Using `isinstance()` checks might prevent an otherwise valid class from working correctly.

The protocol-focused `try:` block might allow a class that coincidentally happens to have a `rank` attribute to work. The risk of this situation turning into a difficult-to-solve problem is nil, as the class would likely fail everywhere else it was used in this application. Also, who compares an instance of `Card` with a class from a financial modeling application that happens to have a rank-ordering attribute?

In future examples, we'll focus on protocol-based comparison using a `try:` block. This tends to offer more flexibility. In cases where flexibility is not desired, the `isinstance()` check can be used.

In our examples, the comparison uses `cast(BlackJackCard, other)` to insist to mypy that the `other` variable conforms to the `BlackjackCard` protocol. In many cases, a

complex class may have a number of protocols defined by various kinds of mixins, and a `cast()` function will focus on the essential mixin, not the overall class.

Comparison methods explicitly return `NotImplemented` to inform Python that this operator isn't implemented for this type of data. Python will try reversing the argument order to see whether the other operand provides an implementation. If no valid operator can be found, then a `TypeError` exception will be raised.

We omitted the three subclass definitions and the factory function, `card21()`. They're left as an exercise.

We also omitted intraclass comparisons; we'll save that for the next section. With this class, we can compare cards successfully. The following is an example where we create and compare three cards:

```
>>> two = card21(2, "♠")
>>> three = card21(3, "♠")
>>> two_c = card21(2, "♣")
```

Given those three `BlackJackCard` instances, we can perform a number of comparisons, as shown in the following code snippet:

```
>>> f"{two} == {three} is {two == three}"
2♠ == 3♠ is False
>>> two.rank == two_c.rank
True
>>> f"{two} < {three} is {two < three}"
2♠ < 3♠ is True
```

The definitions seem to have worked as expected.

Implementation of a comparison of the objects of mixed classes

We'll use the `BlackJackCard` class as an example to see what happens when we attempt comparisons where the two operands are from different classes.

The following is a `Card` instance that we can compare against the `int` values:

```
>>> two = card21(2, "♠")
>>> two < 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Number21Card() < int()
>>> two > 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Number21Card() > int()
```

This is what we expected: the subclass of `BlackJackCard`, `Number21Card`, doesn't provide the required special methods to implement a comparison against integers, so there's a `TypeError` exception. However, consider the following two examples:

```
>>> two == 2
False
>>> 2 == two
False
```

Why do these provide responses? When confronted with a `NotImplemented` value, Python will reverse the operands. In this case, the integer value, `2`, defines the `int.__eq__()` method, which tolerates objects of an unexpected class.

Hard totals, soft totals, and polymorphism

Two classes are polymorphic when they share common attributes and methods. One common example of this is objects of the `int` and `float` classes. Both have `__add__()` methods to implement the `+` operator. Another example of this is that most collections offer a `__len__()` method to implement the `len()` function. The results are produced in different ways, depending on the implementation details.

Let's define `Hand` so that it will perform a meaningful mixed-class comparison among several subclasses of `Hand`. As with same-class comparisons, we have to determine precisely what we're going to compare. We'll look at the following three cases:

- Equality comparisons between `Hand` instances should compare all cards in the collection. Two hands are equal if all of the cards are equal.
- Ordering comparisons between two `Hand` instances should compare an attribute of each `Hand` object. In the case of *Blackjack*, we'll want to compare the hard total or soft total of the hand's points.
- Equality comparisons against an `int` value should compare the `Hand` object's points against the `int` value. In order to have a total, we have to sort out the subtlety of hard totals and soft totals in the game of *Blackjack*.

When there's an ace in a hand, then the following are two candidate totals:

- The **soft total** treats an ace as 11.
- The **hard total** treats an ace as 1. If the soft total is over 21, then only the hard total is relevant to the game.

This means that the hand's total isn't a simple sum of the cards.

We have to determine whether there's an ace in the hand first. Given that information, we can determine whether there's a valid (less than or equal to 21) soft total. Otherwise, we'll fall back on the hard total.

One symptom of **Pretty Poor Polymorphism** is the reliance on `isinstance()` to determine the subclass membership. Generally, this is a violation of the basic ideas of encapsulation and class design. A good set of polymorphic subclass definitions should be completely equivalent with the same method signatures. Ideally, the class definitions are also opaque; we don't need to look inside the class definition. A poor set of polymorphic classes uses extensive `isinstance()` class testing.

In Python, some uses of the `isinstance()` function are necessary. This will arise when using a built-in class. It arises because we can't retroactively add method functions to built-in classes, and it might not be worth the effort of subclassing them to add a polymorphism helper method.

In some of the special methods, it's necessary to use the `isinstance()` function to implement operations that work across multiple classes of objects where there's no simple inheritance hierarchy. We'll show you an idiomatic use of `isinstance()` for unrelated classes in the next section.

For our `cards` class hierarchy, we want a method (or an attribute) that identifies an ace without having to use `isinstance()`. A well-designed method or attribute can help to make a variety of classes properly polymorphic. The idea is to provide a variant attribute value or method implementation that varies based on the class.

We have two general design choices for supporting polymorphism:

- Define a class-level attribute in all relevant classes with a distinct value.
- Define a method in all classes with distinct behavior.

In a situation where the hard total and soft total for the cards differ by 10, this is an indication of at least one ace being present in the hand. We don't need to break encapsulation by checking for class membership. The values of the attributes provide all the information required.

When `card.soft != card.hard`, this is sufficient information to work out the hard total versus the soft total of the hand. Besides indicating the presence of `AceCard`, it also provides the exact offset value between hard and soft totals.

The following is a version of the `total` method that makes use of the soft versus

hard `delta` value:

```
def total(self) -> int:
    delta_soft = max(c.soft - c.hard for c in self.cards)
    hard = sum(c.hard for c in self.cards)
    if hard + delta_soft <= 21:
        return hard + delta_soft
    return hard
```

We'll compute the largest difference between the hard and soft total of each individual card in the hand as `delta_soft`. For most cards, the difference is zero. For an ace, the difference will be nonzero.

Given the hard total and `delta_soft`, we can determine which total to return. If `hard+delta_soft` is less than or equal to `21`, the value is the soft total. If the soft total is greater than 21, then revert to a hard total.

A mixed class comparison example

Given a definition of a total for a `Hand` object, we can meaningfully define comparisons between `Hand` instances and comparisons between `Hand` and `int`. In order to determine which kind of comparison we're doing, we're forced to use `isinstance()`.

The following is a partial definition of `Hand` with comparisons. Here's the first part:

```
class Hand:

    def __init__(self, dealer_card: Card2, *cards: Card2) -> None:
        self.dealer_card = dealer_card
        self.cards = list(cards)

    def __str__(self) -> str:
        return ", ".join(map(str, self.cards))

    def __repr__(self) -> str:
        cards_text = ", ".join(map(repr, self.cards))
        return f"{self.__class__.__name__}({self.dealer_card!r}, {cards_text})"
```

Here's the second part, emphasizing the comparison methods:

```
def __eq__(self, other: Any) -> bool:
    if isinstance(other, int):
        return self.total() == other
    try:
        return (
            self.cards == cast(Hand, other).cards
            and self.dealer_card == cast(Hand, other).dealer_card
        )
    except AttributeError:
        return NotImplemented

def __lt__(self, other: Any) -> bool:
    if isinstance(other, int):
        return self.total() < cast(int, other)
    try:
        return self.total() < cast(Hand, other).total()
    except AttributeError:
        return NotImplemented

def __le__(self, other: Any) -> bool:
    if isinstance(other, int):
        return self.total() <= cast(int, other)
    try:
        return self.total() <= cast(Hand, other).total()
    except AttributeError:
        return NotImplemented
```

```
def total(self) -> int:
    delta_soft = max(c.soft - c.hard for c in self.cards)
    hard = sum(c.hard for c in self.cards)
    if hard + delta_soft <= 21:
        return hard + delta_soft
    return hard
```

We've defined three of the comparisons, not all six. Python's default behavior can fill in the missing operations. Because of the special rules for different types, we'll see that the defaults aren't perfect.

In order to interact with `Hands`, we'll need a few `Card` objects:

```
>>> two = card21(2, '♠')
>>> three = card21(3, '♠')
>>> two_c = card21(2, '♣')
>>> ace = card21(1, '♣')
>>> cards = [ace, two, two_c, three]
```

We'll use this sequence of cards to see two different `hand` instances.

This first `Hands` object has an irrelevant dealer's `Card` object and the set of four `Cards` created previously. One of the `Card` objects is an ace:

```
>>> h = Hand(card21(10,'♠'), *cards)
>>> print(h)
A♣, 2♠, 2♣, 3♠
>>> h.total()
18
```

The total of `18` points is a soft total, because the ace is being treated as having 11 points. The hard total for these cards is 8 points.

The following is a second `Hand` object, which has an additional `Card` object:

```
>>> h2 = Hand(card21(10,'♠'), card21(5,'♠'), *cards)
>>> print(h2)
5♠, A♣, 2♠, 2♣, 3♠
>>> h2.total()
13
```

This hand has a total of `13` points. This is a hard total. The soft total would be over 21, and therefore irrelevant for play.

Comparisons between `Hands` work very nicely, as shown in the following code snippet:

```
>>> h < h2
False
```

```
>>> h > h2
True
```

These comparisons mean that we can rank `Hands` based on the comparison operators. We can also compare `Hands` with integers, as follows:

```
>>> h == 18
True
>>> h < 19
True
>>> h > 17
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Hand() > int()
```

The comparisons with integers work as long as Python isn't forced to try a fallback. The `h > 17` example shows us what happens when there's no `__gt__()` method. Python checks the reflected operands, and the integer, `17`, doesn't have a proper `__lt__()` method for `Hand` either.

We can add the necessary `__gt__()` and `__ge__()` functions to make `Hand` work properly with integers. The code for these two comparisons is left as an exercise for the reader.

The __del__() method

The `__del__()` method has a rather obscure use case.

The intent is to give an object a chance to do any cleanup or finalization just before the object is removed from memory. This use case is handled much more cleanly by context manager objects and the `with` statement. This is the subject of Chapter 6, *Using Callables and Contexts*. Creating a context is much more predictable than dealing with `__del__()` and the Python garbage collection algorithm.

If a Python object has a related operating system resource, the `__del__()` method is the last chance to cleanly disentangle the resource from the Python application. As examples, a Python object that conceals an open file, a mounted device, or perhaps a child subprocess might all benefit from having the resource released as part of `__del__()` processing.

The `__del__()` method is not invoked at any easy-to-predict time. It's not always invoked when the object is deleted by a `del` statement, nor is it always invoked when an object is deleted because a namespace is being removed. The documentation on the `__del__()` method describes the circumstances as *precarious* and provides this additional note on exception processing—exceptions that occur during their execution are ignored, and a warning is printed to `sys.stderr` instead. See the warning here: https://docs.python.org/3/reference/datamodel.html?highlight=__del__#object.__del__.

For these reasons, a context manager is often preferable to implementing `__del__()`.

The reference count and destruction

For CPython implementation, objects have a reference count. The count is incremented when the object is assigned to a variable and decremented when the variable is removed. When the reference count is zero, the object is no longer needed and can be destroyed. For simple objects, `__del__()` will be invoked and the object will be removed.

For complex objects that have circular references among objects, the reference count might never go to zero and `__del__()` can't be invoked easily. The following is a class that we can use to see what happens:

```
class Noisy:

    def __del__(self) -> None:
        print(f"Removing {id(self)}")
```

We can create (and see the removal of) these objects as follows:

```
>>> x = Noisy()
>>> del x
Removing 4313946640
```

We created and removed a `Noisy` object, and, almost immediately, we saw the message from the `__del__()` method. This indicates that the reference count went to zero when the `x` variable was deleted. Once the variable is gone, there's no longer a reference to the `Noisy` instance and it, too, can be cleaned up. The following is a common situation that involves the shallow copies that are often created:

```
>>> ln = [Noisy(), Noisy()]
>>> ln2= ln[:]
>>> del ln
```

There's no response to this `del` statement. The `Noisy` objects have not had their reference counts go to zero yet; they're still being referenced somewhere, as shown in the following code snippet:

```
>>> del ln2
Removing 4313920336
Removing 4313920208
```

The `ln2` variable was a shallow copy of the `ln` list. The `Noisy` objects were referenced in two lists. The `Noisy` instances could not be destroyed until both lists were removed, reducing the reference counts to zero.

There are numerous other ways to create shallow copies. The following are a few ways to create shallow copies of objects:

```
a = b = Noisy()
c = [Noisy()] * 2
```

The point here is that we can often be confused by the number of references to an object that exist because shallow copies are prevalent in Python.

Circular references and garbage collection

The following is a common situation that involves circularity. One class, `Parent`, contains a collection of children. Each `Child` instance contains a reference to the `Parent` class. We'll use these two classes to examine circular references:

```
class Parent:

    def __init__(self, *children: 'Child') -> None:
        for child in children:
            child.parent = self
        self.children = {c.id: c for c in children}

    def __del__(self) -> None:
        print(
            f"Removing {self.__class__.__name__} {id(self):d}"
        )


class Child:

    def __init__(self, id: str) -> None:
        self.id = id
        self.parent: Parent = cast(Parent, None)

    def __del__(self) -> None:
        print(
            f"Removing {self.__class__.__name__} {id(self):d}"
        )
```

A `Parent` instance has a collection of children in a simple `dict`. Note that the parameter value, `*children`, has a type hint of `'Child'`. The `Child` class has not been defined yet. In order to provide a type hint, mypy will resolve a string to a type that's defined elsewhere in the module. In order to have a forward reference or a circular reference, we have to use strings instead of a yet-to-be-defined type.

Each `Child` instance has a reference to the `Parent` class that contains it. The reference is created during initialization, when the children are inserted into the parent's internal collection.

We've made both classes noisy so that we can see when the objects are removed. The following is what happens:

```
>>> p = Parent(Child('a'), Child('b'))
```

```
>>> del p
```

The `Parent` instance and two initial `Child` instances cannot be removed. They both contain references to each other. Prior to the `del` statement, there are three references to the `Parent` object. The `p` variable has one reference. Each `child` object also has a reference. When the `del` statement removed the `p` variable, this decremented the reference count for the `Parent` instance. The count is not zero, so the object remains in memory, unusable. We call this a **memory leak**.

We can create a childless `Parent` instance, as shown in the following code snippet:

```
>>> p_0 = Parent()
>>> id(p_0)
4313921744
>>> del p_0
Removing Parent 4313921744
```

This object is deleted immediately, as expected.

Because of the mutual or circular references, a `Parent` instance and its list of `Child` instances cannot be removed from the memory. If we import the garbage collector interface, `gc`, we can collect and display these nonremovable objects.

We'll use the `gc.collect()` method to collect all the nonremovable objects that have a `__del__()` method, as shown in the following code snippet:

```
>>> import gc
>>> gc.collect()
Removing Child 4536680176
Removing Child 4536680232
Removing Parent 4536679952
30
```

We can see that our `Parent` object is cleaned up by the manual use of the garbage collector. The `collect()` function locates objects that are inaccessible, identifies any circular references, and forces their deletion.

Note that we can't break the circularity by putting code in the `__del__()` method. The `__del__()` method is called *after* the circularity has been broken and the reference counts are already zero. When we have circular references, we can no longer rely on simple Python reference counting to clear out the memory of unused objects. We must either explicitly break the circularity or use a `weakref` reference, which permits garbage collection.

Circular references and the weakref module

In cases where we need circular references but also want `__del__()` to work nicely, we can use **weak references**. One common use case for circular references is mutual references: a parent with a collection of children where each child has a reference back to the parent. If a `Player` class has multiple hands, it might be helpful for a `Hand` object to contain a weak reference to the owning `Player` class.

The default object references could be called **strong references**; however, **direct references** is a better term. They're used by the reference-counting mechanism in Python; they cannot be ignored.

Consider the following statement:

```
a = B()
```

The `a` variable has a direct reference to the object of the `B` class that was created. The reference count to the instance of `B` is at least one, because the `a` variable has a reference.

A weak reference involves a two-step process to find the associated object. A weak reference will use `x.parent()`, invoking the weak reference as a callable object to track down the actual parent object. This two-step process allows the reference counting or garbage collection to remove the referenced object, leaving the weak reference dangling.

The `weakref` module defines a number of collections that use weak references instead of strong references. This allows us to create dictionaries that, for example, permit the garbage collection of otherwise unused objects.

We can modify our `Parent` and `Child` classes to use weak references from `Child` to `Parent`, permitting simpler destruction of unused objects.

The following is a modified class that uses weak references from `Child` to `Parent`:

```
from weakref import ref


class Parent2:

    def __init__(self, *children: 'Child2') -> None:
        for child in children:
            child.parent = ref(self)
        self.children = {c.id: c for c in children}

    def __del__(self) -> None:
        print(
            f"Removing {self.__class__.__name__} {id(self):d}"
        )

class Child2:

    def __init__(self, id: str) -> None:
        self.id = id
        self.parent: ref[Parent2] = cast(ref[Parent2], None)

    def __del__(self) -> None:
        print(
            f"Removing {self.__class__.__name__} {id(self):d}"
        )
```

We've changed the `child` to `parent` reference to be a `weakref` object reference instead of a simple, direct reference.

From within a `Child` class, we must locate the `parent` object via a two-step operation:

```
p = self.parent()
if p is not None:
    # Process p, the Parent instance.
else:
    # The Parent instance was garbage collected.
```

We should explicitly check to be sure the referenced object was found. Objects with weak references can be removed, leaving the weak reference *dangling* – it not longer refers to an object in memory. There are several responses, which we'll look at below.

When we use this new `Parent2` class, we see that `del` makes the reference counts go to zero, and the object is immediately removed:

```
>>> p = Parent2(Child(), Child())
>>> del p
Removing Parent2 4303253584
Removing Child 4303256464
Removing Child 4303043344
```

When a `weakref` reference is dangling (because the referent was destroyed), we

have several potential responses:

- Recreate the referent. You could, perhaps, reload it from a database.
- Use the `warnings` module to write the debugging information in low-memory situations where the garbage collector removed objects unexpectedly and try to continue in degraded mode.
- Ignore it.

Generally, `weakref` references are left dangling after objects have been removed for very good reasons: variables have gone out of scope, a namespace is no longer in use, or the application is shutting down. For these kinds of reasons, the third response is quite common. The object trying to create the reference is probably about to be removed as well.

The __del__() and close() methods

The most common use for `__del__()` is to ensure that files are closed.

Generally, class definitions that open files will have something like what's shown in the following code:

```
__del__ = close
```

This will ensure that the `__del__()` method is also the `close()` method. When the object is no longer needed, the file will be closed, and any operating system resources can be released.

Anything more complex than this is better done with a context manager. See Chapter 6, *Using Callables and Contexts*, for more information on context managers.

The __new__() method and immutable objects

One use case for the `__new__()` method is to initialize objects that are otherwise immutable. The `__new__()` method is where an uninitialized object is created prior to the `__init__()` method setting the attribute values of the object.

The `__new__()` method must be overridden to extend an immutable class where the `__init__()` method isn't used.

The following is a class that does not work. We'll define a version of `float` that carries around information on units:

```
class Float_Fail(float):

    def __init__(self, value: float, unit: str) -> None:
        super().__init__(value)
        self.unit = unit
```

We're trying (improperly) to initialize an immutable object. Since immutable objects can't have their state changed, the `__init__()` method isn't meaningful and isn't used.

The following is what happens when we try to use this class definition:

```
>>> s2 = Float_Fail(6.5, "knots")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
 TypeError: float expected at most 1 arguments, got 2
```

From this, we can see that we can't override the `__init__()` method for the built-in immutable `float` class. We have similar problems with all other immutable classes. We can't set the attribute values on the immutable object, `self`, because that would break the definition of immutability. We can only set attribute values during the object construction. The `__new__()` method supports this kind of processing.

The `__new__()` method is a class method: it will receive the `class` object as the first argument value. This is true without using the `@classmethod` decorator. It doesn't

use a `self` variable, as its job is to create the object that will eventually be assigned to the `self` variable.

For any class we define, the default implementation of `__new__()` is inherited from the parent class. Implicitly, the `class` object is the parent of all classes. The `object.__new__()` method builds a simple, empty object of the required class. The arguments and keywords to `__new__()`, with the exception of the `cls` argument, are passed to `__init__()` as part of the standard Python behavior.

The following are two cases when this default behavior isn't perfect:

- When we want to subclass an immutable class definition. We'll look at this next.
- When we need to create a metaclass. That's the subject of the next section, as it's fundamentally different from creating immutable objects.

Instead of overriding `__init__()` when creating a subclass of a built-in immutable type, we have to tweak the object at the time of creation by overriding `__new__()`. The following is an example class definition that shows us the proper way to extend `float`:

```
class Float_Units(float):

    def __new__(cls, value, unit):
        obj = super().__new__(cls, float(value))
        obj.unit = unit
        return obj
```

This implementation of `__new__()` does two things. It creates a new `Float_Units` object with a float value. It also injects an additional `unit` attribute into the instance that is being created.

It's difficult to provide appropriate type hints for this use of `__new__()`. The method as defined in the typeshed used by mypy version 0.630 doesn't correspond precisely to the underlying implementation. For this rare case, type hints don't seem helpful for preventing problems.

The following code snippet gives us a floating-point value with attached unit information:

```
>>> speed = Float_Units(6.8, "knots")
>>> speed*2
```

```
13.6
>>> speed.unit
'knots'
```

Note that an expression such as `speed * 2` does not create a `Float_Units` object. This class definition inherits all the operator special methods from `float`; the `float` arithmetic special methods all create `float` objects. Creating `Float_Units` objects will be covered in `Chapter 8`, *Creating Numbers*.

The __new__() method and metaclasses

The other use case for the `__new__()` method is to create a metaclass to control how a class definition is built. This use of `__new__()` to build a `class` object is related to using `__new__()` to build a new immutable object, shown previously. In both cases, `__new__()` gives us a chance to make minor modifications in situations where `__init__()` isn't relevant.

A metaclass is used to build a class. Once a `class` object has been built, the `class` object is used to build `instance` objects. The metaclass of all class definitions is `type`. The `type()` function creates the `class` objects in an application. Additionally, the `type()` function can be used to reveal the class of an object.

The following is a silly example of building a new, nearly useless class directly with `type()` as a constructor:

```
Useless = type("Useless", (), {})
```

To create a new class, the `type()` function is given a string name for the class, a tuple of superclasses, and a dictionary used to initialize any `class` variables. The return value is a `class` value. Once we've created this class, we can create objects of this `Useless` class. However, the objects won't do much because they have no methods or attributes.

We can use this newly-minted `Useless` class to create objects, for what little it's worth. The following is an example:

```
>>> Useless = type("Useless", (), {})
>>> u = Useless()
>>> u.attribute = 1
>>> dir(u)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '
```

This example created an instance of `Useless`, `u`. It's easy to add an attribute to the objects of this class with an assignment to `u.attribute`.

This is almost equivalent to defining minimal classes, as follows:

```
from types import SimpleNamespace
Useless2 = SimpleNamespace

class Useless3: pass
```

The definition of `Useless2` is the `SimpleNamespace` class from the `types` module. The definition of `Useless3` uses Python syntax to create a class that's the default implementation of `object`. These all have nearly identical behaviors.

This brings up the important question: why would we change the way classes are defined in the first place?

The answer is that some of the default features of a class aren't perfectly applicable to some edge cases. We'll talk about three situations where we might want to introduce a metaclass:

- We can use a metaclass to add attributes or methods to a class. Note that we're adding these to the class itself, not to any of the instances of the class. The reason for using a metaclass is to simplify the creation of a large number of similar classes. In many respects, adding a `@classmethod` decorator to a method can be similar to creating a metaclass.
- Metaclasses are used to create **Abstract Base Classes** (**ABC**), which we'll look at in Chapter 4, *Attribute Access, Properties, and Descriptions* through Chapter 7, *Creating Containers and Collections*. An ABC relies on a metaclass `__new__()` method to confirm that the concrete subclass is complete. We'll introduce this in Chapter 5, *The ABCs of Consistent Design*.
- Metaclasses can be used to simplify some aspects of object serialization. We'll look at this in Chapter 10, *Serializing and Saving - JSON, YAML, Pickle, CSV, and XML*. When a large number of classes will all be using similar serialization techniques, a metaclass can ensure that all of the application classes have a common serialization aspect.

In general, there are a great many things that can be done in a metaclass that cannot be understood by the mypy tool. It's not always helpful to struggle with the details of defining metaclasses.

Metaclass example – class-level logger

When we have a large number of classes that all need a logger, it can be handy to centralize the feature in a single definition. There are a number of ways of doing this, one of which is to provide a metaclass definition that builds a class-level logger shared by all instances of the class.

The recipe has the following three parts:

1. Create a metaclass. The `__new__()` method of the metaclass will add attributes to the constructed class definition.
2. Create an abstract superclass that is based on the metaclass. This abstract class will simplify inheritance for the application classes.
3. Create the subclasses of the abstract superclass that benefit from the metaclass.

The following is an example metaclass, which will inject a logger into a class definition:

```
import logging

class LoggedMeta(type):

    def __new__(
            cls: Type,
            name: str,
            bases: Tuple[Type, ...],
            namespace: Dict[str, Any]
    ) -> 'Logged':
        result = cast('Logged', super().__new__(cls, name, bases, namespace))
        result.logger = logging.getLogger(name)
        return result

class Logged(metaclass=LoggedMeta):
    logger: logging.Logger
```

The `LoggedMeta` class extends the built-in default metaclass, `type`, with a new version of the `__new__()` method.

The `__new__()` metaclass method is executed after the class body elements have been added to the namespace. The argument values are the metaclass, the new class name to be built, a tuple of superclasses, and a namespace with all of the class items used to initialize the new class. This example is typical: it uses `super()`

to delegate the real work of `__new__()` to the superclass. The superclass of this metaclass is the built-in `type` class.

The `__new__()` method in this example also adds an attribute, `logger`, into the class definition. This was not provided when the class was written, but will be available to every class that uses this metaclass.

We must use the metaclass when defining a new abstract superclass, `Logged`. Note that the superclass includes a reference to the `logger` attribute, which will be injected by the metaclass. This information is essential to make the injected attribute visible to mypy.

We can then use this new abstract class as the superclass for any new classes that we define, as follows:

```
class SomeApplicationClass(Logged):
    def __init__(self, v1: int, v2: int) -> None:
        self.logger.info("v1=%r, v2=%r", v1, v2)
        self.v1 = v1
        self.v2 = v2
        self.v3 = v1*v2
        self.logger.info("product=%r", self.v3)
```

The `__init__()` method of the `SomeApplication` class relies on the `logger` attribute available in the class definition. The `logger` attribute was added by the metaclass, with a name based on the class name. No additional initialization or setup overhead is required to ensure that all the subclasses of `Logged` have `loggers` available.

Summary

We've looked at a number of basic special methods, which are essential features of any class that we design. These methods are already part of every class, but the defaults that we inherit from the object may not match our processing requirements.

We'll almost always need to override `__repr__()`, `__str__()`, and `__format__()`. The default implementations of these methods aren't very helpful at all.

We rarely need to override `__bool__()` unless we're writing our own collection. That's the subject of `Chapter 7`, *Creating Containers and Collections*.

We often need to override comparison and `__hash__()` methods. These definitions are suitable for simple immutable objects, but are not at all appropriate for mutable objects. We may not need to write all the comparison operators; we'll look at the `@functools.total_ordering` decorator in `Chapter 9`, *Decorators and Mixins - Cross-Cutting Aspects*.

The other two basic special method names, `__new__()` and `__del__()`, are for more specialized purposes. Using `__new__()` to extend an immutable class is the most common use case for this method function.

These basic special methods, along with `__init__()`, will appear in almost every class definition that we write. The rest of the special methods are for more specialized purposes; they fall into six discrete categories:

- **Attribute access**: These special methods implement what we see as `object.attribute` in an expression, `object.attribute` on the left-hand side of an assignment, and `object.attribute` in a `del` statement.
- **Callables**: A special method implements what we see as a function applied to arguments, much like the built-in `len()` function.
- **Collections**: These special methods implement the numerous features of collections. This involves operations such as `sequence[index]`, `mapping[key]`, and `set | set`.
- **Numbers**: These special methods provide the arithmetic operators and the

comparison operators. We can use these methods to expand the domain of numbers that Python works with.

- **Contexts**: There are two special methods that we'll use to implement a context manager that works with the `with` statement.
- **Iterators**: There are special methods that define an iterator. This isn't essential, as generator functions handle this feature so elegantly. However, we'll look at how we can design our own iterators.

In the next chapter, we will address attributes, properties, and descriptors.

Attribute Access, Properties, and Descriptors

An object is a collection of features, including methods and attributes. The default behavior of the `object` class involves setting, getting, and deleting named attributes. We often need to modify this behavior to change the attributes available in an object.

This chapter will focus on the following five tiers of attribute access:

- We'll look at built-in attribute processing.
- We'll review the `@property` decorator. A property extends the concept of an attribute to include the processing defined in method functions.
- We'll look at how to make use of the lower-level special methods that control attribute access: `__getattr__()`, `__setattr__()`, and `__delattr__()`. These special methods allow us to build more sophisticated attribute processing.
- We'll also take a look at the `__getattribute__()` method, which provides more granular control over attributes. This can allow us to write very unusual attribute handling.
- We'll take a look at descriptors. These objects mediate access to an attribute. Therefore, they involve somewhat more complex design decisions. Descriptors are the foundational structure used to implement properties, static methods, and class methods.

In this chapter, we'll see how the default processing works in detail. This will help us to decide where and when to override the default behavior. In some cases, we want our attributes to do more than simply be instance variables. In other cases, we might want to prevent the addition of attributes. We may have attributes that have even more complex behaviors.

Also, as we explore descriptors, we'll come to a much deeper understanding of how Python's internals work. We don't often need to use descriptors explicitly. We often use them implicitly, however, because they implement a number of Python features.

Since type hints are now available in Python, we'll take a look at how to annotate attributes and properties so a tool like **mypy** can confirm that objects of appropriate types are used.

Finally, we'll look at the new `dataclasses` module and how this can be used to simplify class definition.

In this chapter, we will cover the following topics:

- Basic attribute processing
- Creating properties
- Using special methods for attribute access
- The `getattribute_()` method
- Creating descriptors
- Using Type Hints for attributes and properties
- Using the `dataclasses` module

Technical requirements

The code files for this chapter can be found at `https://git.io/fj2Uu`.

Basic attribute processing

By default, any class we create will permit the following four behaviors with respect to attributes:

- To create a new attribute and set its value
- To set the value of an existing attribute
- To get the value of an attribute
- To delete an attribute

We can experiment with this using something as simple as the following code. We can create a simple, generic class and an object of that class:

```
>>> class Generic:
...     pass
...
>>> g = Generic()
```

The preceding code permits us to create, get, set, and delete attributes. We can easily create and get an attribute. The following are some examples:

```
>>> g.attribute = "value"
>>> g.attribute
'value'
>>> g.unset
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Generic' object has no attribute 'unset'
>>> del g.attribute
>>> g.attribute
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Generic' object has no attribute 'attribute'
```

The example shows adding, changing, and removing attributes. We will get exceptions if we try to get an otherwise unset attribute or delete an attribute that doesn't exist yet.

A slightly better way to do this is to use an instance of the `types.SimpleNamespace` class. The feature set is the same, but we don't need to create an extra class definition. We create an object of the `SimpleNamespace` class instead, as follows:

```
>>> import types
>>> n = types.SimpleNamespace()
```

In the following code, we can see that the same use cases work for a `SimpleNamespace` class:

```
>>> n.attribute = "value"
>>> n.attribute
'value'
>>> del n.attribute
>>> n.attribute
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'namespace' object has no attribute 'attribute'
```

We can create attributes for this instance, `n`. Any attempt to use an undefined attribute raises an exception.

An instance of the `SimpleNamespace` class has different behavior from what we saw when we created an instance of the `object` class. An instance of the `object` class doesn't permit the creation of new attributes; it lacks the internal `__dict__` structure that Python uses to store attributes and values.

Attributes and the __init__() method

Most of the time, we create an initial collection of attributes using the `__init__()` method of a class. Ideally, we provide names and default values for all the attributes in `__init__()`.

While it's not *required* to provide all attributes in the `__init__()` method, it's the place **mypy** checks to gather the expected list of attributes of an object. An optional attribute can be used as part of an object's state, but there aren't easy ways to describe the absence of an attribute as a valid state for an object.

An optional attribute also pushes the edge of the envelope in terms of class definition. A class is defined by the unique collection of attributes. Attributes are best added (or removed) by creating a subclass or superclass definition. Dynamic changes to the attributes are confusing to tools such as **mypy** as well as to people.

Generally, optional attributes imply a concealed or informal subclass relationship. Therefore, we bump up against Pretty Poor Polymorphism when we use optional attributes. Multiple polymorphic subclasses are often a better implementation than optional attributes.

Consider a *Blackjack* game in which only a single split is permitted. If a hand is split, it cannot be re-split. There are several ways that we can model this constraint:

- We can create an instance of a subclass, `SplitHand`, from the `Hand.split()` method. We won't show this in detail. This subclass of `Hand` has an implementation for `split()` that raises an exception. Once a `Hand` has been split to create two `SplitHand` instances, these cannot be re-split.
- We can create a status attribute on an object named `Hand`, which can be created from the `Hand.split()` method. Ideally, this is a Boolean value, but we can implement it as an optional attribute as well.

The following is a version of `Hand.split()` that can detect splittable versus unsplittable hands via an optional attribute, `self.split_blocker`:

```
def  split(self, deck):
    assert self.cards[0].rank == self.cards[1].rank
    try:
        self.split_blocker
        raise CannotResplit
    except AttributeError:
        h0 = Hand(self.dealer_card, self.cards[0], deck.pop())
        h1 = Hand(self.dealer_card, self.cards[1], deck.pop())
        h0.split_blocker = h1.split_blocker = True
        return h0, h1
```

The `split()` method tests for the presence of a `split_blocker` attribute. If this attribute exists, then this hand should not be re-split; the method raises a customized `CannotSplit` exception. If the `split_blocker` attribute does not exist, then splitting is allowed. Each of the resulting objects has the optional attribute, preventing further splits.

An optional attribute has the advantage of leaving the `__init__()` method relatively uncluttered with status flags. It has the disadvantage of obscuring an aspect of object state. Furthermore, the **mypy** program will be baffled by the reference to an attribute not initialized in `__init__()`. Optional attributes for managing object state must be used carefully, if at all.

Creating properties

A property is a method function that appears (syntactically) to be a simple attribute. We can get, set, and delete property values with syntax identical to the syntax for attribute values. There's an important distinction, though. A property is actually a method and can process, rather than simply preserve, a reference to another object.

Besides the level of sophistication, one other difference between properties and attributes is that we can't attach new properties to an existing object easily, but we can add dynamic attributes to an object very easily. A property is not identical to a simple attribute in this one respect.

There are two ways to create properties. We can use the `@property` decorator, or we can use the `property()` function. The differences are purely syntactic. We'll focus on the decorator.

We'll now take a look at two basic design patterns for properties:

- **Eager calculation**: In this design pattern, when we set a value via a property, other attributes are also computed.
- **Lazy calculation**: In this design pattern, calculations are deferred until requested via a property.

In order to compare the preceding two approaches to properties, we'll split some common features of the `Hand` object into an abstract superclass, as follows:

```
class Hand:

    def __init__(
            self,
            dealer_card: BlackJackCard,
            *cards: BlackJackCard
        ) -> None:
        self.dealer_card = dealer_card
        self._cards = list(cards)

    def __str__(self) -> str:
        return ", ".join(map(str, self.card))

    def __repr__(self) -> str:
        return (
            f"{self.__class__.__name__}"
```

```
            f"({self.dealer_card!r}, "
            f"{', '.join(map(repr, self.card))})"
        )
```

In the preceding code, we defined the object initialization method, which actually does nothing. There are two string representation methods provided. This class is a wrapper around an internal list of cards, kept in an instance variable, `_cards`. We've used a leading `_` on the instance variable as a reminder that this is an implementation detail that may change.

The `__init__()` is used to provide instance variable names and type hints for **mypy**. An attempt to use `None` as a default in this kind of abstract class definition will violate the type hints. The `dealer_card` attribute must be an instance of `BlackJackCard`. To allow this variable to have an initial value of `None`, the type hint would have to be `Optional[BlackJackCard]`, and all references to this variable would also require a guarding `if` statement to be sure the value was not `None`.

The following is a subclass of `Hand`, where `total` is a lazy property that is computed only when needed:

```
class Hand_Lazy(Hand):

    @property
    def total(self) -> int:
        delta_soft = max(c.soft - c.hard for c in self._cards)
        hard_total = sum(c.hard for c in self._cards)
        if hard_total + delta_soft <= 21:
            return hard_total + delta_soft
        return hard_total

    @property
    def card(self) -> List[BlackJackCard]:
        return self._cards

    @card.setter
    def card(self, aCard: BlackJackCard) -> None:
        self._cards.append(aCard)

    @card.deleter
    def card(self) -> None:
        self._cards.pop(-1)
```

The `Hand_Lazy` class sets the `dealer_card` and the `_cards` instance variables. The `total` property is based on a method that computes the total only when requested. Additionally, we defined some other properties to update the collection of cards in the hand. The `card` property can get, set, or delete cards in the hand. We'll take a look at these properties in the `setter` and `deleter` properties section.

We can create a `Hand_Lazy` object. `total` appears to be a simple attribute:

```
>>> d = Deck()
>>> h = Hand_Lazy(d.pop(), d.pop(), d.pop())
>>> h.total
19
>>> h.card = d.pop()
>>> h.total
29
```

The total is computed lazily by rescanning the cards in the hand each time the total is requested. For the simple `BlackJackCard` instances, this is a relatively inexpensive computation. For other kinds of items, this can involve considerable overhead.

Eagerly computed properties

The following is a subclass of `Hand`, where `total` is a simple attribute that's computed eagerly as each card is added:

```
class Hand_Eager(Hand):

    def __init__(
            self,
            dealer_card: BlackJackCard,
            *cards: BlackJackCard
    ) -> None:
        self.dealer_card = dealer_card
        self.total = 0
        self._delta_soft = 0
        self._hard_total = 0
        self._cards: List[BlackJackCard] = list()
        for c in cards:
            # Mypy cannot discern the type of the setter.
            # https://github.com/python/mypy/issues/4167
            self.card = c  # type: ignore

    @property
    def card(self) -> List[BlackJackCard]:
        return self._cards

    @card.setter
    def card(self, aCard: BlackJackCard) -> None:
        self._cards.append(aCard)
        self._delta_soft = max(aCard.soft - aCard.hard, self._delta_soft)
        self._hard_total = self._hard_total + aCard.hard
        self._set_total()

    @card.deleter
    def card(self) -> None:
        removed = self._cards.pop(-1)
        self._hard_total -= removed.hard
        # Issue: was this the only ace?
        self._delta_soft = max(c.soft - c.hard for c in self._cards)
        self._set_total()

    def _set_total(self) -> None:
        if self._hard_total + self._delta_soft <= 21:
            self.total = self._hard_total + self._delta_soft
        else:
            self.total = self._hard_total
```

The `__init__()` method of the `Hand_Eager` class initializes the eagerly computed `total` to zero. It also uses two other instance variables, `_delta_soft`, and `_hard_total`, to track the state of ace cards in the hand. As each card is placed in the hand, these totals are updated.

Each use of `self.card` looks like an attribute. It's actually a reference to the property method decorated with `@card.setter`. This method's parameter, `aCard`, will be the value on the right side of the `=` in an assignment statement.

In this case, each time a card is added via the card `setter` property, the `total` attribute is updated.

The other `card` property decorated with `@card.deleter` eagerly updates the `total` attribute whenever a card is removed. We'll take a look at `deleter` in detail in the next section.

A client sees the same syntax between these two subclasses (`Hand_Lazy()` and `Hand_Eager()`) of `Hand`:

```
d = Deck()
h1 = Hand_Lazy(d.pop(), d.pop(), d.pop())
print(h1.total)
h2 = Hand_Eager(d.pop(), d.pop(), d.pop())
print(h2.total)
```

In both cases, the client software simply uses the `total` attribute. The lazy implementation defers computation of totals until required, but recomputes them every time. The eager implementation computes totals immediately, and only recomputes them on a change to the hand. The trade-off is an important software engineering question, and the final choice depends on how the overall application uses the `total` attribute.

The advantage of using properties is that the syntax doesn't change when the implementation changes. We can make a similar claim for `getter`/`setter` method functions. However, `getter`/`setter` method functions involve extra syntax that isn't very helpful or informative. The following are two examples, one of which involves the use of a `setter` method, while the other uses the assignment operator:

```
obj.set_something(value)
obj.something = value
```

The presence of the assignment operator (`=`) makes the intent very plain. Many programmers find it clearer to look for assignment statements than to look for `setter` method functions.

The setter and deleter properties

In the previous examples, we defined the `card` property to deal additional cards into an object of the `Hand` class.

Since `setter` (and `deleter`) properties are created from the `getter` property, we must always define a `getter` property first using code that looks like the following example:

```
@property
def card(self) -> List[BlackJackCard]:
    return self._cards

@card.setter
def card(self, aCard: BlackJackCard) -> None:
    self._cards.append(aCard)

@card.deleter
def card(self) -> None:
    self._cards.pop(-1)
```

This allows us to add a card to the hand with a simple statement like the following:

```
h.card = d.pop()
```

The preceding assignment statement has a disadvantage because it looks like it replaces all the cards with a single card. On the other hand, it has an advantage in that it uses simple assignment to update the state of a mutable object. We can use the `__iadd__()` special method to do this a little more cleanly. However, we shall wait until Chapter 8, *Creating Numbers*, to introduce the other special methods.

We will consider a version of `split()` that works like the following code:

```
def split(self, deck: Deck) -> "Hand":
    """Updates this hand and also returns the new hand."""
    assert self._cards[0].rank == self._cards[1].rank
    c1 = self._cards[-1]
    del self.card
    self.card = deck.pop()
    h_new = self.__class__(self.dealer_card, c1, deck.pop())
    return h_new
```

The preceding method updates the given `Hand` instance and returns a new `Hand` object. Because this method is inside the `Hand` class definition, it must show the class name as a string because the class has not been fully defined yet.

The `del` statement will remove the last card from the current hand. This will use the `@card.deleter` property to do the work of deleting the card. For a lazy hand, nothing more needs to be done. For an eager hand, the totals must be updated. The assignment statement in front of the `del` statement was used to save the last card into a local variable, `c1`.

The following is an example of a hand being split:

```
>>> d = Deck()
>>> c = d.pop()
>>> h = Hand_Lazy(d.pop(), c, c)  # Create a splittable hand
>>> h2 = h.split(d)
>>> print(h)
2♠, 10♠
>>> print(h2)
2♠, A♠
```

Once we have two cards, we can use `split()` to produce the second hand. A card was removed from the initial hand to create the resulting hand.

This version of `split()` is certainly workable. However, it seems somewhat better to have the `split()` method return two fresh new `Hand` objects. That way, the old, pre-split `Hand` instance can be saved as a memento for auditing or statistical analysis.

Using special methods for attribute access

We'll now look at the three canonical special methods for attribute access: `__getattr__()`, `__setattr__()`, and `__delattr__()`. Additionally, we'll acknowledge the `__dir__()` special method to reveal attribute names. We'll defer `__getattribute__()` to the next section.

The default behavior shown in this section is as follows:

- The `__setattr__()` method will create and set attributes.
- The `__getattr__()` method is a fallback used when an attribute is not defined. When an attribute name is not part of the instance variables of an object, then the `__getattr__()` method is used. The default behavior of this special method is to raise an `AttributeError` exception. We can override this to return a meaningful result instead of raising an exception.
- The `__delattr__()` method deletes an attribute.
- The `__dir__()` method returns a list of attribute names. This is often coupled with `__getattr__( )` to provide a seamless interface to attributes computed dynamically.

The `__getattr__()` method function is only one step in a larger process; it is used when the attribute name is unknown. If the name is a known attribute, this method is not used.

We have a number of design choices for controlling attribute access. Some of these design choices are as follows:

- We can replace the internal `__dict__` with `__slots__`. This makes it difficult to add new attributes. The named attributes remain mutable, however.
- We can use the two special methods to add attributes to a class by overriding `__setattr__()` and `__delattr__()`. Dynamic attributes make it difficult for **mypy** to evaluate type hints.
- We can implement property-like behaviors in a class. Using `__getattr__()` and `__setattr__()` methods, we can ensure that a variety of property-like

processing is centralized in these two methods.

- We can create lazy attributes where the values aren't (or can't be) computed until they're needed. For example, we can create an attribute that doesn't have a value until it's read from a file, database, or network. This is a common use for `__getattr__()`.
- We can have eager attributes, where setting an attribute creates values in other attributes immediately. This is done via overrides to `__setattr__()`.

We won't look at all of these alternatives. Instead, we'll focus on the most commonly used techniques. We'll create objects with a limited number of attributes and look at other ways to compute dynamic attribute values.

When we are not able to set an attribute or create a new attribute, then the object is immutable. The following is what we'd like to see in interactive Python:

```
>>> c = card21(1,'♠')
>>> c.rank = 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 30, in __setattr__
TypeError: Cannot set rank
>>> c.hack = 13
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 31, in __setattr__
AttributeError: 'Ace21Card' has no attribute 'hack'
```

The preceding code shows a `Card` object, where we are not allowed to change an attribute or add an attribute to the object.

The simplest way to implement completely immutable behavior is to extend `typing.NamedTuple`. We'll look at this in the sections that follow. It is the preferred approach. Prior to that, we'll look at some more complex alternatives for selective features of immutability.

Limiting attribute names with __slots__

We can use `__slots__` to create a class where we cannot add new attributes, but can modify an attribute's value. This example shows how to restrict the attribute names:

```
class BlackJackCard:
    __slots__ = ("rank", "suit", "hard", "soft")

    def __init__(self, rank: str, suit: "Suit", hard: int, soft: int) -> None:
        self.rank = rank
        self.suit = suit
        self.hard = hard
        self.soft = soft
```

We made one significant change to the previous definitions of this class: setting the `__slots__` attribute to the names of the attributes allowed. This turns off the internal `__dict__` feature of the object and limits us to these attribute names only. The defined attribute values are mutable even though new attributes cannot be added.

The primary use case for this feature is to limit the memory occupied by the internal `__dict__` structure created by default. The `__slots__` structure uses less memory, and is often used when a very large number of instances will be created.

Dynamic attributes with __getattr__()

We can create objects where attributes are computed from a single, centralized `__getattr__()` method. When attribute values are computed by separate properties, the presence of many methods can be a convenient way to encapsulate a variety of algorithms. In some cases, however, it might be sensible to combine all of the computations into a single method. In this case, the names of the attributes are essentially invisible to **mypy**, since they aren't an obvious part of the Python source text.

A single computation method is shown in the following example:

```
class RTD_Solver:

    def __init__(
        self, *,
        rate: float = None,
        time: float = None,
        distance: float = None
    ) -> None:
        if rate:
            self.rate = rate
        if time:
            self.time = time
        if distance:
            self.distance = distance

    def __getattr__(self, name: str) -> float:
        if name == "rate":
            return self.distance / self.time
        elif name == "time":
            return self.distance / self.rate
        elif name == "distance":
            return self.rate * self.time
        else:
            raise AttributeError(f"Can't compute {name}")
```

An instance of the `RTD_Solver` class is built with two of three values. The idea is to compute the missing third value from the other two. In this case, we've elected to make the missing value an optional attribute, and compute the value of the attribute when required. The attributes for this class are dynamic: two of the three possible attributes will be in use.

The class is used as shown in the following snippet:

```
>>> r1 = RTD_Solver(rate=6.25, distance=10.25)
```

```
>>> r1.time
1.64
>>> r1.rate
6.25
```

An instance of the `RTD_Solver` class was built with two of the three possible attributes. In this example, it's `rate` and `distance`. A request for the `time` attribute value leads to a computation of time from rate and distance.

A request for the rate attribute value, however, does not involve the `__getattr__()` method. Because the instance has `rate` and `distance` attributes, these are provided directly. To confirm that `__getattr__()` is not used, insert a `print()` function in the computation of rate, as shown in the following code snippet:

```
if name == "rate":
    print("Computing Rate")
    return self.distance / self.time
```

When an instance of `RTD_Solver` is created with an attribute value set by the `__init__()` method, the `__getattr__()` method is not used to fetch the attribute. The `__getattr__()` method is only used for unknown attributes.

Creating immutable objects as a NamedTuple subclass

The best approach in terms of creating immutable objects is to make our `Card` property a subclass of `typing.NamedTuple`.

The following is an extension to the built-in `typing.NamedTuple` class:

```
class AceCard2(NamedTuple):
    rank: str
    suit: Suit
    hard: int = 1
    soft: int = 11

    def __str__(self) -> str:
        return f"{self.rank}{self.suit}"
```

When we use the preceding code, we see the following kinds of interaction:

```
>>> c = AceCard2("A", Suit.Spade)
>>> c.rank
'A'
>>> c.suit
<Suit.Spade: '♠'>
>>> c.hard
1
```

We can create an instance, and it has the desired attribute values. We cannot, however, add or change any attributes. All of the processing of attribute names is handled by the `NamedTuple` class definition:

```
>>> c.not_allowed = 2
Traceback (most recent call last):
  File "/Users/slott/miniconda3/envs/py37/lib/python3.7/doctest.py", line 1329, in __rur
    compileflags, 1), test.globs)
  File "<doctest __main__.__test__.test_comparisons_2[3]>", line 1, in <module>
    c.not_allowed = 2
AttributeError: 'AceCard2' object has no attribute 'not_allowed'
>>> c.rank = 3
Traceback (most recent call last):
  File "/Users/slott/miniconda3/envs/py37/lib/python3.7/doctest.py", line 1329, in __rur
    compileflags, 1), test.globs)
  File "<doctest __main__.__test__.test_comparisons_2[4]>", line 1, in <module>
    c.rank = 3
AttributeError: can't set attribute
```

Eagerly computed attributes, dataclasses, and __post_init__()

We can define an object where attributes are computed eagerly — as soon as — possible after a value is set. An object such as this can optimize access by performing a computation once and leaving the result to be used multiple times.

This can be done with property setters. However, a class with a lot of property setters, each of which computes a number of attributes, can be rather complex-looking. In some cases, all of the derived value computations can be centralized.

The `dataclasses` module provides us with a class with an array of built-in features. One of these features is a `__post_init__()` method that we can use to derive values eagerly.

We'd like something that looks like the following code:

```
>>> RateTimeDistance(rate=5.2, time=9.5)
RateTimeDistance(rate=5.2, time=9.5, distance=49.4)
>>> RateTimeDistance(distance=48.5, rate=6.1)
RateTimeDistance(rate=6.1, time=7.950819672131148, distance=48.5)
```

We can set two of the three required values in this `RateTimeDistance` object. The additional attribute is computed immediately, as demonstrated in the following code block:

```
from dataclasses import dataclass

@dataclass
class RateTimeDistance:

    rate: Optional[float] = None
    time: Optional[float] = None
    distance: Optional[float] = None

    def __post_init__(self) -> None:
        if self.rate is not None and self.time is not None:
            self.distance = self.rate * self.time
        elif self.rate is not None and self.distance is not None:
            self.time = self.distance / self.rate
        elif self.time is not None and self.distance is not None:
            self.rate = self.distance / self.time
```

A class defined by the `@dataclass` decorator will accept a variety of initialization values. After the values have been set, the `__post_init__()` method is invoked. This can be used to compute additional values.

The attributes here are mutable, and it's relatively simple to create an object with inconsistent values for rate, time, and distance. We can do the following to create an object with improper internal relationships among the attribute values:

```
>>> r1 = RateTimeDistance(time=1, rate=0)
>>> r1.distance = -99
```

To prevent this, a `@dataclass(frozen=True)` decorator can be used. This variant will behave quite a bit like a `NamedTuple`.

Incremental computation with __setattr__()

We can create classes which use `__setattr__()` to detect changes in attribute values. This can lead to incremental computation. The idea is to build derived values after initial attribute values have been set.

Note the complexity of having two senses of attribute setting.

- The client's view: An attribute can be set and other derived values may be computed. In this case, a sophisticated `__setattr__()` is used.
- The internal view: Setting an attribute must not result in any additional computation. If additional computation is done, this leads to an infinite recursion of setting attributes and computing derived values from those attributes. In this case, the fundamental `__setattr__()` method of the superclass must be used.

This distinction is important and easy to overlook. Here's a class that both sets attributes and computes derived attributes in the `__setattr__()` method:

```
class RTD_Dynamic:
    def __init__(self) -> None:
        self.rate : float
        self.time : float
        self.distance : float

        super().__setattr__('rate', None)
        super().__setattr__('time', None)
        super().__setattr__('distance', None)

    def __repr__(self) -> str:
        clauses = []
        if self.rate:
            clauses.append(f"rate={self.rate}")
        if self.time:
            clauses.append(f"time={self.time}")
        if self.distance:
            clauses.append(f"distance={self.distance}")
        return (
            f"{self.__class__.__name__}"
            f"({', '.join(clauses)})"
        )

    def __setattr__(self, name: str, value: float) -> None:
        if name == 'rate':
            super().__setattr__('rate', value)
```

```
        elif name == 'time':
            super().__setattr__('time', value)
        elif name == 'distance':
            super().__setattr__('distance', value)

        if self.rate and self.time:
            super().__setattr__('distance', self.rate * self.time)
        elif self.rate and self.distance:
            super().__setattr__('time', self.distance / self.rate)
        elif self.time and self.distance:
            super().__setattr__('rate', self.distance / self.time)
```

The `__init__()` method uses the `__setattr__()` superclass to set default attribute values without starting a recursive computation. The instance variables are named with type hints, but no assignment is performed.

The `RTD_Dynamic` class provides a `__setattr__()` method that will set an attribute. If enough values are present, it will also compute derived values. The internal use of `super().__setattr__()` specifically avoids any additional computations from being done by using the `object` superclass attribute setting methods.

Here's an example of using this class:

```
>>> rtd = RTD_Dynamic()
>>> rtd.time = 9.5
>>> rtd
RTD_Dynamic(time=9.5)
>>> rtd.rate = 6.25
>>> rtd
RTD_Dynamic(rate=6.25, time=9.5, distance=59.375)
>>> rtd.distance
59.375
```

Note that we can't set attribute values inside some methods of this class using simple `self.name =` *syntax.*

Let's imagine we tried to write the following line of code inside the `__setattr__()` method of this class definition:

```
self.distance = self.rate*self.time
```

If we were to write the preceding code snippet, we'd have infinite recursion in the `__setattr__()` method. In the `self.distance=x` line, this is implemented as `self.__setattr__('distance', x)`. If a line such as `self.distance=x` occurs within the body of `__setattr__()`, it means `__setattr__()` will have to be used while trying to implement attribute settings. The `__setattr__()` superclass doesn't do any additional work and is free from recursive entanglements with itself.

It's also important to note that once all three values are set, changing an attribute

won't simply recompute the other two attributes. The rules for computation are based on an explicit assumption that one attribute is missing and the other two are available.

To properly recompute values, we need to make two changes: 1) set the desired attribute to `None`, and 2) provide a value to force a recomputation.

We can't simply set a new value for `rate` and compute a new value for `time` while leaving `distance` unchanged. To tweak this model, we need to both clear one variable and set a new value for another variable:

```
>>> rtd.time = None
>>> rtd.rate = 6.125
>>> rtd
RTD_Dynamic(rate=6.125, time=9.5, distance=58.1875)
```

Here, we cleared `time` and changed `rate` to get a new solution for `time` using the previously established value for `distance`.

The __getattribute__() method

An even lower-level attribute processing is the `__getattribute__()` method. The default implementation of this method attempts to locate the value as an existing attribute in the internal `__dict__` (or `__slots__`). If the attribute is not found, this method calls `__getattr__()` as a fallback. If the value located is a descriptor (refer to the following *Creating descriptors* section), then it processes the descriptor. Otherwise, the value is simply returned.

By overriding this method, we can perform any of the following kinds of tasks:

- We can effectively prevent access to attributes. This method, by raising an exception instead of returning a value, can make an attribute more secret than if we were to merely use the leading underscore (`_`) to mark a name as private to the implementation.
- We can invent new attributes similarly to how `__getattr__()` can invent new attributes. In this case, however, we can bypass the default lookup done by the default version of `__getattribute__()`.
- We can make attributes perform unique and different tasks. This might make the program very difficult to understand or maintain, and it could also be a terrible idea.
- We can change the way descriptors behave. While technically possible, changing a descriptor's behavior sounds like a terrible idea.

When we implement the `__getattribute__()` method, it's important to note that there cannot be any internal attribute references in the method's body. If we attempt to get the value for `self.name`, it will lead to infinite recursion of the `__getattribute__()` method.

The `__getattribute__()` method cannot use any simple `self.name` attribute access; it will lead to infinite recursions.

In order to get attribute values within the `__getattribute__()` method, we must explicitly refer to the base method defined in a superclass, or the base class `object`, as shown in the following snippet:

```
object.__getattribute__(self, name)
```

We can use this kind of processing to inject debugging, audit, or security controls into a class definition. We might, for example, write a line to a log when an attribute is accessed in a particularly important class. A sensible security test might limit access to people with defined access controls.

The following example will show a trivial use of `__getattribute__()` to prevent access to the single leading `_` instance variables and methods in a class. We'll do this by raising an `AttributeError` exception for any of those kinds of names.

Here's the class definition:

```
class SuperSecret:

    def __init__(self, hidden: Any, exposed: Any) -> None:
        self._hidden = hidden
        self.exposed = exposed

    def __getattribute__(self, item: str):
        if (len(item) >= 2 and item[0] == "_"
                and item[1] != "_"):
            raise AttributeError(item)
        return super().__getattribute__(item)
```

We've overridden `__getattribute__()` to raise an attribute error on private names only. This will leave Python's internal `__` names visible, but any name with a single `_` prefix will be concealed. The `_hidden` attribute will be nearly invisible. The following is an example of an object of this class being used:

```
>>> x = SuperSecret('onething', 'another')
>>> x.exposed
'another'
>>> x._hidden  # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
  File "/Users/slott/miniconda3/envs/py37/lib/python3.7/doctest.py", line 1329, in __rur
    compileflags, 1), test.globs)
  File "<doctest __main__.__test__.test_secret[3]>", line 1, in <module>
    x._hidden  #
  File "/Users/slott/Documents/Writing/Python/Mastering OO Python 2e/mastering-oo-pythor
    raise AttributeError(item)
AttributeError: _hidden
```

The object, `x`, will respond to requests for the exposed attribute, but will raise an exception for any reference to an attribute that begins with `_`.

This does not fully conceal all of the `_` names, however. The `dir()` function will show the existence of the `_hidden` attribute. To correct this problem, the `__dir__()` special method must be overridden to also conceal the names beginning with one `_`.

As general advice, it's rarely necessary to change the implementation of `__getattribute__()`. The default implementation gives us access to flexible features via property definitions or as changes to `__getattr__()`.

Creating descriptors

A descriptor is a class that mediates attribute access. The descriptor class can be used to get, set, or delete attribute values. Descriptor objects are built inside a class at class definition time. Descriptors are the essence of how Python implements methods, attributes, and properties.

The descriptor design pattern has two parts: an **owner class** and the **attribute descriptor** itself. The owner class uses one or more descriptors for its attributes. A descriptor class defines some combination of the `__get__`, `__set__`, and `__delete__` methods. An instance of the descriptor class will be an attribute of the owner class.

A descriptor is an instance of a class that is separate from the owning class. Therefore, descriptors let us create reusable, generic kinds of attributes. The owning class can have multiple instances of each descriptor class to manage attributes with similar behaviors.

Unlike other attributes, descriptors are created at the class level. They're not created within the `__init__()` initialization. While descriptor instances can have values set during initialization, the descriptor instances are generally built as part of the class, outside any method functions. Each descriptor object will be an instance of a descriptor class. The descriptor instance must be bound to an attribute name in the owner class.

To be recognized as a descriptor, a class must implement any combination of the following three methods:

- `Descriptor.__get__(self, instance, owner)`: In this method, the `instance` parameter is the `self` variable of the object being accessed. The `owner` parameter is the owning class object. If this descriptor is invoked in a class context, the `instance` parameter will get a `None` value. This must return the value of the descriptor.
- `Descriptor.__set__(self, instance, value)`: In this method, the `instance` parameter is the `self` variable of the object being accessed. The `value` parameter is the new value that the descriptor needs to be set to.

- `Descriptor.__delete__(self, instance)`: In this method, the `instance` parameter is the `self` variable of the object being accessed. This method of the descriptor must delete this attribute's value.

Sometimes, a descriptor class will also need an `__init__()` method function to initialize the descriptor's internal state. There are two design patterns for descriptors based on the methods defined, as follows:

- **A non-data descriptor**: This kind of descriptor defines only the `__get__()` method. The idea of a non-data descriptor is to provide an indirect reference to another object via methods or attributes of its own. A non-data descriptor can also take some action when referenced.
- **A data descriptor**: This descriptor defines both `__get__()` and `__set__()` to create a mutable object. It may also define `__delete__()`. A reference to an attribute with a value of a data descriptor is delegated to the `__get__()`, `__set__()`, or `__delete__()` methods of the descriptor object.

There are a wide variety of use cases for descriptors. Internally, Python uses descriptors for several reasons:

- The methods of a class are implemented as descriptors. These are non-data descriptors that apply the method function to the object and the various parameter values.
- The `property()` function is implemented by creating a data descriptor for the named attribute.
- A class method, or static method, is implemented as a descriptor. In both cases, the method will apply to the class instead of an instance of the class.

When we look at object-relational mapping in Chapter 12, *Storing and Retrieving Objects via SQLite*, we'll see that many of the ORM class definitions make use of descriptors to map Python class definitions to SQL tables and columns.

As we think about the purposes of a descriptor, we must also examine the three common use cases for the data that a descriptor works with, as follows:

- The **descriptor object** has, or acquires, the data value. In this case, the descriptor object's `self` variable is relevant, and the descriptor object is stateful. With a data descriptor, the `__get__()` method can return this internal data. With a non-data descriptor, the descriptor may include other methods

or attributes to acquire or process data. Any descriptor state applies to the class as a whole.

- The **owner instance** contains the data. In this case, the descriptor object must use the `instance` parameter to reference a value in the owning object. With a data descriptor, the `__get__()` method fetches the data from the instance. With a non-data descriptor, the descriptor's other methods access the instance data.
- The **owner class** contains the relevant data. In this case, the descriptor object must use the `owner` parameter. This is commonly used when the descriptor implements a static method or class method that applies to the class as a whole.

We'll take a look at the first case in detail. This means creating a data descriptor with `__get__()` and `__set__()` methods. We'll also look at creating a non-data descriptor without a `__get__()` method.

The second case (the data in the owning instance) is essentially what the `@property` decorator does. There's a small possible advantage to writing a descriptor class instead of creating a conventional property—a descriptor can be used to refactor the calculations into the descriptor class. While this may fragment class design, it can help when the calculations are truly of epic complexity. This is essentially the **Strategy** design pattern, a separate class that embodies a particular algorithm.

The third case shows how the `@staticmethod` and `@classmethod` decorators are implemented. We don't need to reinvent those wheels.

Using a non-data descriptor

Internally, Python uses non-data descriptors as part of the implementation for class methods and static methods. This is possible because a descriptor provides access to the owning class, as well as the instance.

We'll look at an example of a descriptor that updates the instance and also works with the filesystem to provide an additional side-effect to use of the descriptor.

For this example, we'll add a descriptor to a class that will create a working directory that is unique to each instance of a class. This can be used to cache state, or debugging history, or even audit information in a complex application.

Here's an example of an abstract class that might use a `StateManager` internally:

```
class PersistentState:
    """Abstract superclass to use a StateManager object"""
    _saved: Path
```

The `PersistentState` class definition includes a reference to an attribute, `_saved`, which has a type hint of `Path`. This formalizes the relationships among the objects in a way that can be detected by **mypy**.

Here's an example of a descriptor that provides access to a file for saving the object state:

```
class StateManager:
    """May create a directory. Sets _saved in the instance."""

    def __init__(self, base: Path) -> None:
        self.base = base

    def __get__(self, instance: PersistentState, owner: Type) -> Path:
        if not hasattr(instance, "_saved"):
            class_path = self.base / owner.__name__
            class_path.mkdir(exist_ok=True, parents=True)
            instance._saved = class_path / str(id(instance))
        return instance._saved
```

When this descriptor is created in a class, a base `Path` is provided. When this instance is referenced, it will ensure that a working directory exists. It will also save a working `Path` object, setting the `_saved` instance attribute.

The following is a class that uses this descriptor for access to a working directory:

```
class PersistentClass(PersistentState):
    state_path = StateManager(Path.cwd() / "data" / "state")

    def __init__(self, a: int, b: float) -> None:
        self.a = a
        self.b = b
        self.c: Optional[float] = None
        self.state_path.write_text(repr(vars(self)))

    def calculate(self, c: float) -> float:
        self.c = c
        self.state_path.write_text(repr(vars(self)))
        return self.a * self.b + self.c

    def __str__(self) -> str:
        return self.state_path.read_text()
```

At the class level, a single instance of this descriptor is created. It's assigned to the `state_path` attribute. There are three places where a reference to `self.state_path` are made. Because the object is a descriptor, the `__get__()` method is invoked implicitly each time the variable is referenced. This means that any of those references will serve to create the necessary directory and working file path.

This implicit use of the `__get__()` method of the `StateManager` class will guarantee consistent processing at each reference. The idea is to centralize the OS-level work into a single method that is part of a reusable descriptor class.

As an aid to debugging, the `__str__()` method dumps the content of the file into which the state has been written. When we interact with this class, we see output like the following example:

```
>>> x = PersistentClass(1, 2)
>>> str(x)
"{'a': 1, 'b': 2, 'c': None, '_saved': ...)}"
>>> x.calculate(3)
5
>>> str(x)
"{'a': 1, 'b': 2, 'c': 3, '_saved': ...)}"
```

We created an instance of the `PersistentClass` class, providing initial values for two attributes, `a`, and `b`. The third attribute, `c`, is left with a default value of `None`. The use of `str()` displays the content of the saved state file.

The reference to `self.saved_state` invoked the descriptor's `__get__()` method,

ensuring that the directory exists and could be written.

This example demonstrates the essential feature of a non-data descriptor. The implied use of the `__get__()` method can be handy for performing a few, limited kinds of automated processing where implementation details need to be hidden. In the case of static methods and class methods, this is very helpful.

Using a data descriptor

A data descriptor is used to build property-like processing using external class definitions. The descriptor methods of `__get__()`, `__set__()`, and `__delete__()` correspond to the way `@property` can be used to build `getter`, `setter`, and `deleter` methods. The important distinction of the descriptor is a separate and reusable class definition, allowing reuse of property definitions.

We'll design an overly simplistic unit conversion schema using descriptors that can perform appropriate conversions in their `__get__()` and `__set__()` methods.

The following is a superclass of a descriptor of units that will do conversions to and from a standard unit:

```
class Conversion:
    """Depends on a standard value."""
    conversion: float
    standard: str

    def __get__(self, instance: Any, owner: type) -> float:
        return getattr(instance, self.standard) * self.conversion

    def __set__(self, instance: Any, value: float) -> None:
        setattr(instance, self.standard, value / self.conversion)

class Standard(Conversion):
    """Defines a standard value."""
    conversion = 1.0
```

The `Conversion` class does simple multiplications and divisions to convert standard units to other non-standard units, and vice versa. This doesn't work for temperature conversions, and a subclass is required to handle that case.

The `Standard` class is an extension to the `Conversion` class that sets a standard value for a given measurement without any conversion factor being applied. This exists mostly to provide a very visible name to the standard for any particular kind of measurement.

With these two superclasses, we can define some conversions from a standard unit. We'll look at the measurement of speed. Some concrete descriptor class definitions are as follows:

```
class Speed(Conversion):
    standard = "standard_speed"  # KPH


class KPH(Standard, Speed):
    pass


class Knots(Speed):
    conversion = 0.5399568


class MPH(Speed):
    conversion = 0.62137119
```

The abstract `Speed` class provides the standard source data for the various conversion subclasses, `KPH`, `Knots`, and `MPH`. Any attributes based on subclasses of the `Speed` class will consume standard values.

The `KPH` class is defined as a subclass of both `Standard` class and the `Speed` class. From `Standard`, it gets a conversion factor of 1.0. From `Speed`, it gets the attribute name to be used to keep the standard value for speed measurements.

The other classes are subclasses of `Speed`, which performs conversions from a standard value to the desired value.

The following `Trip` class uses these conversions for a given measurement:

```
class Trip:
    kph = KPH()
    knots = Knots()
    mph = MPH()

    def __init__(
        self,
        distance: float,
        kph: Optional[float] = None,
        mph: Optional[float] = None,
        knots: Optional[float] = None,
    ) -> None:
        self.distance = distance  # Nautical Miles
        if kph:
            self.kph = kph
        elif mph:
            self.mph = mph
        elif knots:
            self.knots = knots
        else:
            raise TypeError("Impossible arguments")
        self.time = self.distance / self.knots

    def __str__(self) -> str:
        return (
            f"distance: {self.distance} nm, "
            f"rate: {self.kph} "
```

```
            f"kph = {self.mph} "
            f"mph = {self.knots} knots, "
            f"time = {self.time} hrs"
        )
```

Each of the class-level attributes, `kph`, `knots`, and `mph`, are descriptors for a different unit. When these attributes are referenced, the `__get__()` and `__set__()` methods of the various descriptors will perform appropriate conversions to and from the standard values.

The following is an example of an interaction with the `Trip` class:

```
>>> m2 = Trip(distance=13.2, knots=5.9)
>>> print(m2)
distance: 13.2 nm, rate: 10.92680006993152 kph = 6.789598762345432 mph = 5.9 knots, time
>>> print(f"Speed: {m2.mph:.3f} mph")
Speed: 6.790 mph
>>> m2.standard_speed
10.92680006993152
```

We created an object of the `Trip` class by setting an attribute, `distance`, setting one of the available descriptors, and then computing a derived value, `time`. In this example, we set the `knots` descriptor. This is a subclass of the `Speed` class, which is a subclass of the `Conversion` class, and therefore, the value will be converted to a standard value.

When we displayed the value as a large string, each of the descriptors' `__get__()` methods were used. These methods fetched the internal `kph` attribute value from the owning object, applied a conversion factor, and returned the resulting values.

The process of creating the descriptors allows for reuse of the essential unit definitions. The calculations can be stated exactly once, and they are separate from any particular application class definition. Compare this with a `@property` method that is tightly bound to the class including it. The various conversion factors, similarly, are stated once, and can be widely reused by a number of related applications.

The core description, conversion, embodies a relatively simple computation. When the computation is more complex, it can lead to a sweeping simplification of the overall application. Descriptors are very popular when working with databases and data serialization problems because the descriptor's code can involve complex conversions to different representations.

Using type hints for attributes and properties

When using **mypy**, we'll need to provide type hints for the attributes of a class. This is generally handled through the `__init__()` method. Most of the time, the parameter type hints are all that's required.

In previous examples, we defined classes like this:

```
class RTD_Solver:

    def __init__(
        self, *,
        rate: Optional[float] = None,
        time: Optional[float] = None,
        distance: Optional[float] = None
    ) -> None:
        if rate:
            self.rate = rate
        if time:
            self.time = time
        if distance:
            self.distance = distance
```

The type hints on the parameters are used to discern the types for the instance variables, `self.rate`, `self.time`, and `self.distance`.

When we assign default values in the `__init__()` method, we have two common design patterns.

- When we can compute a value eagerly, the type can be discerned by **mypy** from the assignment statement.
- When a default `None` value is provided, the type will have to be stated explicitly.

We may see assignment statements such as the following:

```
self.computed_value: Optional[float] = None
```

This assignment statement tells **mypy** that the variable will either be an instance of `float` or the `None` object. This style of initialization makes the class attribute types explicit.

For property definitions, the type hint is part of the property method definition. We'll often see code like the following:

```
@property
def some_computed_value(self) -> float: ...
```

This definition provides a clear statement for the type of `object.some_computed_value`. This is used by **mypy** to be sure the types all match among the references to this property name.

Using the dataclasses module

Starting with Python 3.7 the `dataclasses` module is available. This module offers a superclass we can use to create classes with clearly-stated attribute definitions. The core use case for a dataclass is a simple definition of the attributes of a class.

The attributes are used to automatically create common attribute access methods, including `__init__()`, `__repr__()`, and `__eq__()`. Here's an example:

```
from dataclasses import dataclass
from typing import Optional, cast

@dataclass
class RTD:
    rate: Optional[float]
    time: Optional[float]
    distance: Optional[float]

    def compute(self) -> "RTD":
        if (
            self.distance is None and self.rate is not None
            and self.time is not None
        ):
            self.distance = self.rate * self.time
        elif (
            self.rate is None and self.distance is not None
            and self.time is not None
        ):
            self.rate = self.distance / self.time
        elif (
            self.time is None and self.distance is not None
            and self.rate is not None
        ):
            self.time = self.distance / self.rate
        return self
```

Each instance of this class will have three attributes, `rate`, `time`, and `distance`. The decorator will create an `__init__()` method to set these attributes. It will also create a `__repr__()` method to display the details of the attribute value. An `__eq__()` method is written to perform a simple equality check on all of the attribute values.

Careful checking for `None` and non-`None` values is helpful for **mypy**. This explicit checking provides an assurance that the `Optional[float]` types will have non-`None` values.

Note that the three names are written as part of the class definition. They're used to build an `__init__()` method that's part of the resulting class. These will become instance variables in the resulting objects.

The `compute()` method changes the internal state of the object. We've provided a type hint that describes the return value as an instance of the class. Here's how we can use an instance of this class:

```
>>> r = RTD(distance=13.5, rate=6.1, time=None)
>>> r.compute()
RTD(rate=6.1, time=2.2131147540983607, distance=13.5)
```

In this code snippet, we created an instance, providing non-`None` values for `distance` and `rate`. The `compute()` method computed a value for the `time` attribute.

The default `@dataclass` decorator will not have comparison methods. It will create a mutable class where attribute values can be changed.

We can request some additional, optional features. We can provide optional parameters to the decorator to control optional features. We can create a class for immutable objects with comparison operators with code such as the following:

```
@dataclass(frozen=True, order=True)
class Card:
    rank: int
    suit: str

    @property
    def points(self) -> int:
        return self.rank
```

The `frozen` parameter in this example leads the decorator to make the class into an immutable, *frozen* object. The `order` parameter to the `@dataclass` decorator creates the methods for comparison in the class definition. This is very helpful for creating simple, immutable objects. Because the two attributes include type hints, **mypy** can confirm that the `Card` dataclass is used properly.

Inheritance works with dataclasses. We can declare classes as in the following example:

```
class Ace(Card):

    @property
    def points(self) -> int:
        return 1
```

```
class Face(Card):

    @property
    def points(self) -> int:
        return 10
```

These two classes inherit the `__init__()`, `__repr__()`, `__eq__()`, `__hash__()`, and comparison methods from the `Card` superclass. These two classes differ in the implementation of the `points()` method.

The `@dataclass` decorator simplifies the class definition. The methods that tend to have a direct relationship with the attributes are generated by the decorator.

Attribute Design Patterns

Programmers coming from other languages (particularly Java and C++) can try to make all attributes private and write extensive `getter` and `setter` functions. This kind of design pattern can be necessary for languages where type definitions are statically compiled into the runtime. It is not necessary in Python. Python depends on a different set of common patterns.

In Python, it's common to treat all attributes as public. This means the following:

- All attributes should be well documented.
- Attributes should properly reflect the state of the object; they shouldn't be temporary or transient values.
- In the rare case of an attribute that has a potentially confusing (or brittle) value, a single leading underscore character (`_`) marks the name as *not part of the defined interface*. It's not technically private, but it can't be relied on in the next release of the framework or package.

It's important to think of private attributes as a nuisance. Encapsulation isn't broken by the lack of complex privacy mechanisms in the language; proper encapsulation can only be broken by bad design.

Additionally, we have to choose between an attribute or a property which has the same syntax as an attribute, but can have more complex semantics.

Properties versus attributes

In most cases, attributes can be set outside a class with no adverse consequences. Our example of the `Hand` class shows this. For many versions of the class, we can simply append to `hand.cards`, and the lazy computation of `total` via a property will work perfectly.

In cases where the changing of an attribute should lead to consequential changes in other attributes, a more sophisticated class design is required:

- A method may clarify the state change. This will be necessary when multiple parameter values are required and the changes must be synchronized.
- A `setter` property may be clearer than a method function. This will be a sensible option when a single value is required.
- We can also use Python's in-place operators, such as `+=`. We'll defer this until `Chapter 8`, *Creating Numbers*.

There's no strict rule. The distinction between a method function and a property is entirely one of syntax and how well the syntax communicates the intent. For computed values, a property allows lazy computation, while an attribute requires eager computation. This devolves to a performance question. The benefits of lazy versus eager computation are based on the expected use cases.

Finally, for some very complex cases, we might want to use the underlying Python descriptors.

Designing with descriptors

Many uses of descriptors are already part of Python. We don't need to reinvent properties, class methods, or static methods.

The most compelling cases for creating new descriptors relate to mapping between Python objects and other software outside Python. Object-relational database mapping, for example, requires a great deal of care to ensure that a Python class has the right attributes in the right order to match a SQL table and columns. Also, when mapping to something outside Python, a descriptor class can handle the encoding and decoding of data, or fetching the data from external sources.

When building a web service client, we might consider using descriptors to make web service requests. The `__get__()` method, for example, might turn into an HTTP `GET` request, and the `__set__()` method might turn into an HTTP `PUT` request. In some cases, a single request may populate the data of several descriptors. In this case, the `__get__()` method would check the instance cache and return that value before making an HTTP request.

Many data descriptor operations are more simply handled by properties. This provides us with a place to start to write properties first. If the property processing becomes too expansive or complex, then we can switch to descriptors to refactor the class.

Summary

In this chapter, we looked at several ways to work with an object's attributes. We can use the built-in features of the `object` class to get and set attribute values simply and effectively. We can use `@property` to create attribute-like methods.

If we want more sophistication, we can tweak the underlying special method implementations for `__getattr__()`, `__setattr__()`, `__delattr__()`, or `__getattribute__()`. These allow us very fine-grained control over attribute behaviors. We walk a fine line when we touch these methods because we can make fundamental (and confusing) changes to Python's behavior.

Internally, Python uses descriptors to implement features such as class methods, static methods, and properties. Many of the really good use cases for descriptors are already first-class features of the language.

The use of type hints helps confirm that objects are used properly. They're strongly encouraged as a supplement to unit tests for assuring that parameters and values align.

The new `dataclasses` module can help simplify class definition. In many cases, a class created with the @dataclass decorator can be the essence of well-designed software.

In the next chapter, we'll look closely at the **ABC**s (**Abstract Base Classes**) that we'll exploit in `Chapter 6`, *Using Callables and Contexts*, `Chapter 7`, *Creating Containers and Collections*, and `Chapter 8`, *Creating Numbers*. These ABCs will help us to define classes that integrate nicely with existing Python features. They will also allow us to create class hierarchies that enforce consistent design and extension.

The ABCs of Consistent Design

The Python standard library provides abstract base classes for a number of container features. It provides a consistent framework for the built-in container classes, such as `list`, `dict`, and `set`. Additionally, the standard library provides abstract base classes for numbers. We can use these classes to extend the suite of numeric classes available in Python.

In this chapter, we'll look in general at the abstract base classes in the `collections.abc` module. From there, we can focus on a few use cases that will be the subject of detailed examination in future chapters.

There are three common design strategies for reusing existing classes: wrap, extend, and invent. We'll look at the general concepts behind the various containers and collections that we might want to wrap or extend. Similarly, we'll look at the concepts behind the numbers that we might want to implement.

Our goal is to ensure that our application classes integrate seamlessly with existing Python features. If we create a collection, for example, it's appropriate to have that collection create an iterator by implementing `__iter__()`. A collection that implements `__iter__()` will work seamlessly with a `for` statement.

Technical requirements

The code files for this chapter can be found at `https://git.io/fj2Uz`.

Abstract base classes

The core of the **abstract base class** (**ABC**) definition is defined in a module named `abc`. This contains the required decorators and metaclasses to create abstractions. Other classes rely on these definitions. The `collections.abc` module uses the `abc` module to create abstractions focused on collections. We'll also look at the `numbers` module, because it contains ABCs for numeric types. There are ABCs for I/O in the `io` module, too.

An abstract base class has the following features:

- **Abstract** means that these classes don't contain *all* the method definitions required to work completely. For it to be a useful subclass, we will need to provide some method definitions.
- **Base** means that other classes will use it as a superclass.
- An **abstract class** provides some definitions for methods. Most importantly, the abstract base classes often provide the signatures for the missing methods. A subclass must provide the right methods to create a concrete class that fits the interface defined by the abstract class.

Bear in mind the following when using abstract base classes:

- When we use them to define our classes, they will be consistent with Python's internal classes.
- We can use them to create some common, reusable abstractions that our application extends.
- We can use them to support the proper inspection of a class to determine what it does. This allows better collaboration among library classes and new classes in our applications. It helps to start from the formal definitions of classes that will have similar behavior to other containers or numbers.

If we don't use abstract base classes, we can easily create a class that fails to provide all the features of the abstract base `Sequence` class. This will lead to a class being *almost* a sequence—we sometimes call it *sequence-like*. This can lead to odd inconsistencies and kludgy workarounds for a class that doesn't quite provide all the features of a `Sequence` class.

With an abstract base class, an application's class is guaranteed to have the advertised features of the abstract base class. If it lacks a feature, the presence of an undefined abstract method will make the class unusable for building object instances.

We will use ABCs in several situations, as follows:

- We will use ABCs as superclasses when defining our own classes.
- We will use ABCs within a method to confirm that an operation is possible.
- We will use ABCs within a diagnostic message or exception to indicate why an operation can't work.

For the first use case, we can write modules with code that looks like the following:

```
import collections.abc
class SomeApplicationClass(collections.abc.Sequence):
    pass
```

Our `SomeApplicationClass` is defined as a `Sequence` class. It must then implement the specific methods required by `Sequence`, or we will not be able to create an instance.

For the second use case, we can write methods with code as follows:

```
def some_method(self, other: Iterator):
    assert isinstance(other, collections.abc.Iterator)
```

Our `some_method()` requires the `other` argument to be a subclass of `Iterator`. If the `other` argument can't pass this test, we get an exception.

Instead of the `assert` statement, a common alternative is an `if` statement that raises `TypeError`, which may be more meaningful than `AssertError`. We'll see this in the following section.

For the third use case, we might have something like the following:

```
try:
    some_obj.some_method(another)
except AttributeError:
    warnings.warn(f"{another!r} not an Iterator, found {another.__class__.__bases__!r}")
    raise
```

In this case, we wrote a diagnostic warning that shows the base classes for a given object. This may help debug problems with application design.

Base classes and polymorphism

In this section, we'll flirt with the idea of **pretty poor polymorphism**. Inspection of argument value types is a Python programming practice that should be isolated to a few special cases. Later, when we look at numbers and numeric coercion, we'll learn about cases where the inspection of types is recommended.

Well-done polymorphism follows what is sometimes called the **Liskov substitution principle**. Polymorphic classes can be used interchangeably. Each polymorphic class has the same suite of properties. For more information, visit http://en.wikipedia.org/wiki/Liskov_substitution_principle.

Overusing `isinstance()` to distinguish between the types of arguments can lead to a needlessly complex (and slow) program. Unit testing is a far better way to find programming errors than verbose type inspection in the code.

Method functions with lots of `isinstance()` methods can be a symptom of a poor (or incomplete) design of polymorphic classes. Rather than having type-specific processing outside of a class definition, it's often better to extend or wrap classes to make them more properly polymorphic and encapsulate the type-specific processing within the class definition.

One potential use of the `isinstance()` method is to raise diagnostic errors. A simple approach is to use the `assert` statement, as follows:

```
assert isinstance(some_argument, collections.abc.Container),
f"{some_argument!r} not a Container"
```

This will raise an `AssertionError` exception to indicate that there's a problem. This has the advantage that it is short and to the point. This example has two disadvantages: assertions can be silenced, and it would probably be better to raise a `TypeError` for this. The preceding use of the `assert` statement is not very helpful, and should be avoided.

The following example is slightly better:

```
if not isinstance(some_argument, collections.abc.Container):
    raise TypeError(f"{some_argument!r} not a Container")
```

The preceding code has the advantage that it raises the correct error. However, it has the disadvantages of being long-winded and it creates a needless constraint on the domain of objects. Objects that are not proper subclasses of the abstract `Container` class may still offer the required methods, and should not be excluded.

The Pythonic approach is summarized as follows:

"It's better to ask for forgiveness than to ask for permission."

This is generally taken to mean that we should minimize the upfront testing of arguments (asking permission) to see if they're the correct type. Argument-type inspections are rarely of any tangible benefit. Instead, we should handle the exceptions appropriately (asking forgiveness).

Checking types in advance is often called **look before you leap** (**LBYL**) programming. It's an overhead of relatively little value. The alternative is called **easier to ask for forgiveness than permission** (**EAFP**) programming, and relies on `try` statements to recover from problems.

What's best is to combine diagnostic information with the exception in the unlikely event that an inappropriate type is used and somehow passed through unit testing into operation.

The following is generally the best approach:

```
try:
    found = value in some_argument
except TypeError:
    if not isinstance(some_argument, collections.abc.Container):
        warnings.warn(f"{some_argument!r} not a Container")
    raise
```

The assignment statement to create the `found` variable assumes that `some_argument` is a proper instance of a `collections.abc.Container` class and will respond to the `in` operator.

In the unlikely event that someone changes the application and `some_argument` is of a class that can't use the `in` operator, the application will write a diagnostic warning message and crash with a `TypeError` exception.

Many classes work with the `in` operator. Trying to wrap this in LBYL

`if` statements may exclude a perfectly workable class. Using the EAFP style allows *any* class to be used that implements the `in` operator.

Callable

Python's definition of a **callable object** includes the obvious function definitions created with the `def` statement.

The `Callable` type hint is used to describe the `__call__()` method, a common protocol in Python. We can see several examples of this in *Python 3 Object-Oriented Programming*, by Dusty Phillips, from Packt Publishing.

When we look at any Python function, we see the following behavior:

```
>>> def hello(text: str):
...     print(f"hello {text}")

>>> type(hello)
<class 'function'>
>>> from collections.abc import Callable
>>> isinstance(hello, Callable)
True
```

When we create a function, it will fit the abstract base class `Callable`. Every function reports itself as `Callable`. This simplifies the inspection of an argument value and helps write meaningful debugging messages.

We'll take a look at callables in more detail in `Chapter 6`, *Using Callables and Contexts*.

Containers and collections

The `collections` module defines a number of collections above and beyond the built-in container classes. The container classes include `namedtuple()`, `deque`, `ChainMap`, `Counter`, `OrderedDict`, and `defaultdict`. All of these are examples of classes based on ABC definitions.

The following is a quick interaction to show how we can inspect collections to see the methods that they support:

```
>>> isinstance({}, collections.abc.Mapping)
True
>>> isinstance(collections.defaultdict(int), collections.abc.Mapping)
True
```

We can inspect the simple `dict` class to see that it follows the `Mapping` protocol and will support the required methods.

We can inspect a `defaultdict` collection to confirm that it is also part of the `Mapping` class hierarchy.

When creating a new kind of container, we have the following two general approaches:

- Use the `collections.abc` classes to formally inherit behaviors that match existing classes. This will also support **mypy** type hint checking and will also provide some useful default behaviors.
- Rely on type hinting to confirm that the methods match the protocol definitions in the `typing` module. This will only support mypy type hint checking.

It's clearer (and more reliable) to use a proper ABC as the base class for one of our application classes. The additional formality has the following two advantages:

- It advertises what our intention was to people reading (and possibly using or maintaining) our code. When we make a subclass of `collections.abc.Mapping`, we're making a very strong claim about how that class

will behave.

- It creates some diagnostic support. If we somehow fail to implement all of the required methods properly, an exception will be raised when trying to create instances of the abstract base class. If we can't run the unit tests because we can't create instances of an object, then this indicates a serious problem that needs to be fixed.

The entire family tree of built-in containers is reflected in the abstract base classes. Lower-level features include `Container`, `Iterable`, and `Sized`. These are a part of higher-level constructs; they require a few specific methods, particularly `__contains__()`, `__iter__()`, and `__len__()`, respectively.

Higher-level features include the following characteristics:

- `Sequence` and `MutableSequence`: These are the abstractions of the `list` and `tuple` concrete classes. Concrete sequence implementations also include `bytes` and `str`.
- `MutableMapping`: This is the abstraction of `dict`. It extends `Mapping`, but there's no built-in concrete implementation of this.
- `Set` and `MutableSet`: These are the abstractions of the `frozenset` and `set` concrete classes.

This allows us to build new classes or extend existing classes and maintain a clear and formal integration with the rest of Python's built-in features.

We'll look at containers and collections in detail in `Chapter 7`, *Creating Containers and Collections*.

Numbers

When creating new numbers (or extending existing numbers), we turn to the `numbers` module. This module contains the abstract definitions of Python's built-in numeric types. These types form a tall, narrow hierarchy, from the simplest to the most elaborate. In this context, simplicity (and elaborateness) refers to the collection of methods available.

There's an abstract base class named `numbers.Number` that defines all of the numeric and number-like classes. We can see that this is true by looking at interactions like the following one:

```
>>> import numbers
>>> isinstance(42, numbers.Number)
True
>>> 355/113
3.1415929203539825
>>> isinstance(355/113, numbers.Number)
True
```

Clearly, integer and float values are subclasses of the abstract `numbers.Number` class. The subclasses of `Number` include `numbers.Complex`, `numbers.Real`, `numbers.Rational`, and `numbers.Integral`. These definitions are roughly parallel to the mathematical concepts used to define the various classes of numbers.

The `decimal.Decimal` class, however, doesn't fit this hierarchy very well. We can check the relationships using the `issubclass()` method as follows:

```
>>> issubclass(decimal.Decimal, numbers.Number)
True
>>> issubclass(decimal.Decimal, numbers.Integral)
False
>>> issubclass(decimal.Decimal, numbers.Real)
False
>>> issubclass(decimal.Decimal, numbers.Complex)
False
>>> issubclass(decimal.Decimal, numbers.Rational)
False
```

While the `decimal.Decimal` class seems closely aligned with `numbers.Real`, it is not formally a subclass of this type.

For a concrete implementation of `numbers.Rational`, look at the `fractions` module.

We'll look at the various kinds of numbers in detail in `Chapter 8`, *Creating Numbers*.

Some additional abstractions

We'll look at some other interesting ABC classes that are less widely extended. It's not that these abstractions are less widely used: it's more that the concrete implementations rarely need extensions or revisions.

We'll look at the iterator, which is defined by `collections.abc.Iterator`. We'll also look at the unrelated concept of a context manager. This isn't defined with the same formality as other ABC classes. We'll look at this in detail in `Chapter 6`, *Using Callables and Contexts*.

In many cases, we'll create iterators using generator functions and the `yield` statement. We'll use an explicit type hint of `typing.Iterator` for these functions.

The iterator abstraction

Iterator objects are created implicitly when we use an iterable container with a `for` statement. We rarely expect to see the iterator object itself. For the most part, it will be a concealed portion of the implementation of the `for` statement. The few times we do care about the iterator object, we rarely want to extend or revise the class definition.

We can expose the implicit iterators that Python uses through the `iter()` function. We can interact with an iterator in the following way:

```
>>> x = [1, 2, 3]
>>> iter(x)
<list_iterator object at 0x1006e3c50>
>>> x_iter = iter(x)
>>> next(x_iter)
1
>>> next(x_iter)
2
>>> next(x_iter)
3
>>> next(x_iter)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> isinstance(x_iter, collections.abc.Iterator)
True
```

In the preceding code, we created an iterator over a list object, assigning it to the `x_iter` variable. The `next()` function will step through the values in that iterator. This shows how iterator objects are stateful, and the `next()` function both returns a value and updates the internal state.

The final `isinstance()` expression confirmed that this iterator object is an instance of `collections.abc.Iterator`.

Most of the time, we'll work with iterators that have been created by the collection classes themselves; however, when we branch out and build our own collection classes or extend a collection class, we may also need to build a unique iterator. We'll look at iterators in `Chapter 7`, *Creating Containers and Collections*.

Contexts and context managers

A context manager is used with the `with` statement. We work with a context manager when we write something like the following:

```
with function(arg) as context:
    process(context)
```

In the preceding code, `function(arg)` creates the context manager. Once the manager is available, the object can be used as needed. In the example, it's an argument to a function. A context manager class may have methods to perform actions within the scope of the context.

One very commonly used context manager is a file. Any time a file is opened, a context should be used to guarantee that the file will also be properly closed. Consequently, we should almost always use a file in the following way:

```
with open("some file") as the_file:
    process(the_file)
```

At the end of the `with` statement, we're assured that the file will be closed properly. This will release any operating system resources, avoiding resource leaks or incomplete processing when exceptions are raised.

The `contextlib` module provides several tools for building proper context managers. Rather than providing an abstract base class, this library offers decorators, which will transform simple functions into context managers, as well as a `contextlib.ContextDecorator` base class, which can be extended to build a class that is a context manager.

We'll look at context managers in detail in `Chapter 6`, *Using Callables and Contexts*.

The abc and typing modules

The core method of creating ABCs is defined in the `abc` module. This module includes the `ABCMeta` class, which provides several features.

First, the `ABCMeta` class ensures that abstract classes can't be instantiated. When a method uses the `@asbtractmethod` decorator, then a subclass that fails to provide this definition cannot be instantiated. A subclass that provides all of the required definitions for the abstract methods can be instantiated properly.

Second, it provides definitions for `__instancecheck__()` and `__subclasscheck__()`. These special methods implement the `isinstance()` and `issubclass()` built-in functions. They provide the checks to confirm that an object (or a class) belongs to the proper ABC. This includes a cache of subclasses to speed up the testing.

The `abc` module also includes a number of decorators for creating abstract method functions that must be provided by a concrete implementation of the abstract base class. The most important of these is the `@abstractmethod` decorator.

If we wanted to create a new abstract base class, we would use something like the following:

```
from abc import ABCMeta, abstractmethod

class AbstractBettingStrategy(metaclass=ABCMeta):
    @abstractmethod
    def bet(self, hand: Hand) -> int:
        return 1

    @abstractmethod
    def record_win(self, hand: Hand) -> None:
        pass

    @abstractmethod
    def record_loss(self, hand: Hand) -> None:
        pass
```

This class includes `ABCMeta` as its metaclass, making it clear this will be an abstract base class.

This abstraction uses the `abstractmethod` decorator to define three abstract methods. Any concrete subclass must define these in order to be a complete

implementation of the abstract base class. For more complex situations, an abstract base class can define the `__subclasshook__()` method to make more complex tests for the required concrete method definitions.

An example of an abstract subclass of the `AbstractBettingStrategy` class is as follows:

```
class Simple_Broken(AbstractBettingStrategy):
   def bet( self, hand ):
        return 1
```

The preceding code defines an abstract class. An instance can't be built because the class doesn't provide the necessary implementations for all three abstract methods.

The following is what happens when we try to build an instance of this class:

```
>>> simple= Simple_Broken()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Simple_Broken with
 abstract methods record_loss, record_win
```

The error message indicates that the concrete class is incomplete. The following is a better concrete class that passes the completeness test:

```
class Simple(AbstractBettingStrategy):
    def bet(self, hand):
        return 1
    def record_win(self, hand):
        pass
    def record_loss(self, hand):
        pass
```

We can build an instance of this class and use it as part of our simulation. The abstraction forces us to clutter up the implementation with two unused methods. The `bet()` method should be the only *required* abstract method. The other two methods should have already been provided with the default implementation of a single `pass` statement by the abstract base class.

Using the __subclasshook__() method

We can define abstract base classes with complex rules for overrides to create concrete subclasses. This is done by implementing the `__subclasshook__()` method of the abstract base class, as shown in the following code:

```
class AbstractBettingStrategy2(ABC):

    @abstractmethod
    def bet(self, hand: Hand) -> int:
        return 1

    @abstractmethod
    def record_win(self, hand: Hand) -> None:
        pass

    @abstractmethod
    def record_loss(self, hand: Hand) -> None:
        pass

    @classmethod
    def __subclasshook__(cls, subclass: type) -> bool:
        """Validate the class definition is complete."""
        if cls is AbstractBettingStrategy2:
            has_bet = any(hasattr(B, "bet") for B in subclass.__mro__)
            has_record_win =  any(hasattr(B, "record_win") for B in subclass.__mro__)
            has_record_loss = any(hasattr(B, "record_loss") for B in subclass.__mro__)
            if has_bet and has_record_win and has_record_loss:
                return True
        return False
```

This class is an abstract base class, built by extension from the `ABC` superclass. As with the previous example, a number of `@abstractmethod` definitions are provided. Any subclass of this class would be like the previous examples of the `AbstractBettingStrategy` class.

When trying to build an instance of the subclass, the `__subclasshook__()` method is invoked to determine whether the object can be built. In this case, there are three individual checks: `has_bet`, `has_record_win`, and `has_record_loss`. If all three checks pass, then the function returns `True` to permit the object to be built; otherwise, the function returns `False` to prevent it building an instance of an incomplete concrete class.

Using `__subclasshook__()` permits nuanced decision making with regard to the validity of a subclass of an abstract class. It can also lead to confusion because

the obvious rule—that is, implement all `@abstractmethod` methods—isn't in use.

Abstract classes using type hints

We can also do some management of the implementation of concrete methods with type hints and the `typing` module. A concrete class will be checked by mypy to be sure it matches the abstract class type hints. This is not as stringent as the checks made by the `ABCMeta` class, since they don't happen at runtime, but only when mypy is used. We can do this by using `raise NotImplementedError` in the body of an abstract class. This will create a runtime error if the application actually creates an instance of an abstract class.

The concrete subclasses define the methods normally. The presence of type hints means mypy can confirm that the subclass provides a proper definition that matches the superclass type hints. This comparison between type hints is perhaps the most important part of creating concrete subclasses. Consider the following two class definitions:

```
from typing import Tuple, Iterator

class LikeAbstract:
    def aMethod(self, arg: int) -> int:
        raise NotImplementedError

class LikeConcrete(LikeAbstract):
    def aMethod(self, arg1: str, arg2: Tuple[int, int]) -> Iterator[Any]:
        pass
```

The `LikeConcrete` class implementation of the `aMethod()` method is clearly different from the `LikeAbstract` superclass. When we run mypy, we'll see an error message like the following:

```
Chapter_5/ch05_ex1.py:96: error: Signature of "aMethod" incompatible with supertype "Lik
```

This will confirm that the `LikeConcrete` subclass is not a valid implementation of the `aMethod()` method. This technique for creating abstract class definitions via type hinting is a feature of mypy, and can be used in conjunction with the `ABCMeta` class to create a robust library that supports both mypy and runtime checks.

Summary, design considerations, and trade-offs

In this chapter, we looked at the essential ingredients of abstract base classes. We saw a few features of each kind of abstraction.

We also learned that one rule for good class design is to inherit as much as possible. We saw two broad patterns here. We also saw the common exceptions to this rule.

Some application classes don't have behaviors that overlap with internal features of Python. From our Blackjack examples, a `Card` isn't much like a number, a container, an iterator, or a context: it's just a playing card. In this case, we can generally invent a new class because there aren't any built-in features to inherit from.

When we look at `Hand`, however, we can see that a `hand` is clearly a container. As we noted when looking at hand classes in `Chapters 2`, *The __init__() Method*, and `Chapter 3`, *Integrating Seamlessly - Basic Special Methods*, the following are three fundamental design strategies:

- Wrapping an existing container
- Extending an existing container
- Inventing a wholly new kind of container

Most of the time, we'll be wrapping or extending an existing container. This fits with our rule of inheriting as much as possible.

When we extend an existing class, our application class will fit into the class hierarchy neatly. An extension to the built-in `list` is already an instance of `collections.abc.MutableSequence`.

When we wrap an existing class, however, we have to carefully consider which parts of the original interface we want to support and which parts we don't want to support. In our examples in the previous chapters, we only wanted to

expose the `pop()` method from the list object we were wrapping.

Because a wrapper class is not a complete mutable sequence implementation, there are many things it can't do. On the other hand, an extension class participates in a number of use cases that just might turn out to be useful. For example, a `hand` that extends `list` will turn out to be iterable.

If we find that extending a class doesn't meet our requirements, we can resort to building an entirely new collection. The ABC definitions provide a great deal of guidance on what methods are required in order to create a collection that can integrate seamlessly with the rest of the Python universe. We'll look at a detailed example of inventing a collection in `Chapter 7`, *Creating Containers and Collections*.

In most cases, type hints will help us create abstraction classes that constrain aspects of the concrete implementations. The abstract base class definitions are checked when the application executes, introducing overheads that may be undesirable. The mypy checks are made — along with unit tests checks — before an application is used, reducing overheads, and improving confidence in the resulting application.

Looking forward

In the coming chapters, we'll make extensive use of the abstract base classes discussed in this chapter. In `Chapter 6`, *Using Callables and Contexts*, we'll look at the relatively simple features of callables and contexts. In `Chapter 7`, *Creating Containers and Collections*, we'll look at the available containers and collections. We'll also look at how to build a unique, new kind of container in this chapter. Lastly, in `Chapter 8`, *Creating Numbers*, we'll look at the various numeric types and how we can create our own kind of number.

Using Callables and Contexts

The **callable** concept in Python includes a variety of different ways to create functions and objects that behave like functions. We can create callable objects that use **memoization** to maintain a cache of answers, therefore performing very quickly. In some cases, memoization is essential for creating an algorithm that finishes within a reasonable amount of time.

The concept of **context** allows us to create elegant, reliable resource management. The `with` statement defines a context and creates a context manager to control the resources used in that context. Python files are generally context managers; when used in a `with` statement, they are properly closed.

We'll look at several ways to create context managers using the tools in the `contextlib` module. Some other useful abstract base classes are in a separate submodule called `collections.abc`.

We'll show a number of variant designs for callable objects. This will show us why a stateful callable object is sometimes more useful than a simple function. We'll also look at how to use some of the existing Python context managers before we dive in and write our own context manager.

The following concepts will be discussed in this chapter:

- Designing callables
- Improving performance
- Using `functools` for memoization
- Complexities and the callable interface
- Managing contexts and the `with` statement
- Defining the `_enter_()` and `_exit_()` methods
- Context manager as a factory

Technical requirements

The code files for this chapter can be found at `https://git.io/fj2Ug`.

Designing callables

There are two easy and commonly-used ways to create callable objects in Python, which are as follows:

- By using the `def` statement to create a function.
- By creating an instance of a class that implements the `_call()_` method. This can be done by using `collections.abc.Callable` as its base class.

Beyond these two, we can also assign a **lambda** form to a variable. A lambda is a small, anonymous function that consists of exactly one expression. We'd rather not emphasize saving lambdas in a variable, as this leads to the confusing situation where we have a function-like callable that's not defined with a `def` statement.

The following is a simple callable object, `pow1`, created from a class:

```
from typing import Callable
IntExp = Callable[[int, int], int]
class Power1:
    def __call__(self, x: int, n: int) -> int:
        p = 1
        for i in range(n):
            p *= x
        return p
pow1: IntExp = Power1()
```

There are three parts to creating a callable object, as follows:

- The type hint defines the parameters and return values from the resulting callable object. In this example, `Callable[[int, int], int]` defines a function with two integer parameters and an integer result. To save repeating it, a new type name, `IntExp`, is assigned.
- We defined the class with a `__call__()` method. The type signature here matches the `IntExp` type definition.
- We created an instance of the class, `pow1()`. This object is callable and behaves like a function. We've also provided a type hint so that **mypy** can confirm that the callable object will have the proper signature.

The algorithm for computing x^n seems to be inefficient. We'll address that later.

Clearly, the body of the `__call__()` method is so simple that a full class definition isn't really necessary. In order to show the various optimizations, we'll start with this simple callable object rather than mutate a function into a callable object.

We can now use the `pow1()` function just as we'd use any other function. Here's how to use the `pow1()` function in a Python command line:

```
>>> pow1(2, 0)
1
>>> pow1(2, 1)
2
>>> pow1(2, 2)
4
>>> pow1(2, 10)
1024
```

We've evaluated the callable object with various kinds of argument values.

It's not required to make a callable object a subclass of `abc.Callable` when working with `mypy`; however, using the abstract base class can help with debugging.

Consider this flawed definition:

```
class Power2(collections.abc.Callable):
    def __call_( self, x, n ):
        p= 1
        for i in range(n):
            p *= x
        return p
```

The preceding class definition has an error and doesn't meet the definition of the callable abstraction.

The following is what happens when we try to create an instance of this class:

```
>>> pow2: IntExp = Power2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Power2 with abstract
 methods __call__
```

It may not be obvious exactly what went wrong, but we have a fighting chance to debug this. If we hadn't subclassed `collections.abc.Callable`, we'd have a somewhat more mysterious problem to debug.

Here's a version of a broken callable that relies on type hints to detect the problem. This is nearly identical to the correct `Power` class shown previously. The

code that contains a tragic flaw is as follows:

```
class Power3:

    def __call_(self, x: int, n: int) -> int:
        p = 1
        for i in range(n):
            p *= x
        return p
```

When we run mypy, we will see complaints about this code. The expected type of the callable object doesn't match the defined `IntExp` type:

```
# Chapter_6/ch06_ex1.py:68: error: Incompatible types in assignment (expression has type
```

If we ignore the mypy error and try to use this class, we'll see runtime problems. The following is what happens when we try to use `Power3` as a class that doesn't meet the expectations of callables and isn't a subclass of `abc.Callable` either:

```
>>> pow3: IntExp = Power3()
>>> pow3(2, 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Power3' object is not callable
```

This error provides less guidance as to why the `Power3` class definition is flawed. The mypy hint provides some assistance in locating the problem.

Improving performance

We'll look at two performance tweaks for the `Power1` class shown previously.

First, we need to switch to a better algorithm. Then, we will require a better algorithm combined with memoization, which involves a cache; therefore, the function becomes stateful. This is where callable objects shine.

The first modification is to use a **divide-and-conquer** design strategy. The previous version chopped the computation of x^n into $\mathbf{O}(n)$ steps; the loop carried out *n* individual multiplication operations. If we can find a way to split the problem into two equal portions, the problem decomposes into $\mathbf{O}(\log_2 n)$ steps.

For example, `pow1(2,1024)`, the `Power1` callable, performs 1,024 individual multiplication operations. We can optimize this down to 10 multiplications, a significant speedup.

Rather than simply multiplying by a fixed value, we'll use the *fast exponentiation* algorithm. It uses three essential rules for computing, as follows:

- If $n = 1$, then $x^0 = 1$, and the result is simply 1.
- If *n* is odd, $n \bmod 2 = 1$, and the result is $x^n = x^{n-1} \times x$. This involves a recursive computation of x^{n-1}. This does one multiplication. However, $n - 1$is an even number, which can be optimized.
- If *n* is even, $n \bmod 2 = 0$, and the result is $x^n = x^{\frac{n}{2}} \times x^{\frac{n}{2}}$. This involves a recursive computation of $x^{\frac{n}{2}}$. This chops the number of multiplications in half.

The following is these recursive callable object:

```
class Power4:

    def __call__(self, x: int, n: int) -> int:
        if n == 0:
            return 1
        elif n % 2 == 1:
            return self.__call__(x, n - 1) * x
        else:  # n % 2 == 0:
            t = self.__call__(x, n // 2)
            return t * t
```

```
pow4: IntExp = Power4()
```

We applied the three rules to the input value:

- If *n* is zero, we'll return 1.
- If *n* is odd, we'll make a recursive call and return $x^{n-1} \times x$.
- If *n* is even, we'll make a recursive call and return $(x^{\frac{n}{2}})^2$.

The execution time is dramatically faster. We can use the `timeit` module to see the difference in performance. See Chapter 1, *Preliminaries, Tools, and Techniques* for information on using `timeit`. When we compare running `pow1(2,1024)` and `pow4(2,1024)` 10,000 times, we'll see something like 183 seconds for the previous version versus 8 seconds for this version.

The following is how we can gather performance data using `timeit`:

```
import timeit

iterative = timeit.timeit( "pow1(2,1024)","""
class Power1():
    def __call__(self, x: int, n: int) -> int:
        p= 1
        for i in range(n):
            p *= x
        return p

pow1= Power1()
""", number=100_000 )
print("Iterative", iterative)
```

We imported the `timeit` module. The `timeit.timeit()` function will evaluate a given statement in the defined context. In this case, our statement is the `pow1(2,1024)` expression. The context for this statement is the definition of the `pow1()` callable object; this includes the import, class definition, and creation of the `pow1` instance.

Note that we provided `number=100_000` to speed things up. If we had used the default value for the number of iterations, it could have taken almost two minutes.

Using memoization or caching

The idea behind memoization is to cache previous results to avoid recomputing them. We'll use considerably more memory, but we can also dramatically speed up performance by avoiding computation.

An ordinary function doesn't have a place to cache previous results. A function is not expected to be stateful. A callable object, however, can be stateful. It can include a cache of previous results.

The following is a memoized version of our `Power` callable object:

```
class Power5:

    def __init__(self):
        self.memo = {}

    def __call__(self, x: int, n: int) -> int:
        if (x, n) not in self.memo:
            if n == 0:
                self.memo[x, n] = 1
            elif n % 2 == 1:
                self.memo[x, n] = self.__call__(x, n-1) * x
            elif n % 2 == 0:
                t = self.__call__(x, n // 2)
                self.memo[x, n] = t * t
            else:
                raise Exception("Logic Error")
        return self.memo[x, n]

pow5: IntExp = Power5()
```

We revised our algorithm to work with a `self.memo` cache. This is initialized to an empty mapping. In the `__call__()` method, the cache is checked for previously computed answers.

If the parameter values have been requested previously, the cached result is returned and no computation is performed. This is the big speedup that we spoke of earlier.

Otherwise, the parameter values are not present in the cache. In this missing value case,the value of x^n must be computed and saved. The three rules to compute the fast exponent are used to get and put values in the cache. This

assures us that future calculations will be able to exploit the cached values.

The importance of memoization can't be stressed enough. The reduction in computation can be dramatic. It is commonly done by replacing a slow, expensive function with a callable object.

Memoization doesn't work well with float values. The lack of exact match equality means some kind of approximately-equal test needs to be made against the cached values. When working with float values, either rounding needs to be used, or some more sophisticated cache search will be required.

Using functools for memoization

The Python library includes a memoization decorator in the `functools` module. We can use this module instead of creating our own callable object.

We can use this as follows:

```
from functools import lru_cache

@lru_cache()
def pow6(x: int, n: int) -> int:
    if n == 0:
        return 1
    elif n % 2 == 1:
        return pow6(x, n-1) * x
    else:  # n % 2 == 0:
        t = pow6(x, n // 2)
        return t * t
```

This code defines a function, `pow6()`, which is decorated with a **Least Recently Used** (**LRU**) cache. Previous requests are stored in a memoization cache. The idea behind an LRU cache is that the most recently made requests are kept and the oldest requests are quietly purged. We can use `@lru_cache(256)`, for example, to limit the cache to 256 entries, thereby optimizing memory use.

Using `timeit`, we can see that 10,000 iterations of `pow5()` run in about 1 second, while the iterations for `pow6()` run in about 8 seconds.

What this also shows is that a trivial use of `timeit` can misstate the performance of the memoization algorithms. If each request is recomputing a previously cached answer, only the first iteration – with an empty cache – performs the computation.

Aiming for simplicity using a callable interface

The idea behind a callable object is that we have a `class` interface focused on a single method. This is also true for simple function definitions.

Some objects have multiple relevant methods. A Blackjack `Hand`, for example, has to add cards and produce a total. A Blackjack `Player` has to place bets, accept hands, and make play decisions (for example, hit, stand, split, insure, and double down). These are more complex interfaces that are not suitable to be callables.

The betting strategy, however, is a candidate for being a callable. While it will be implemented as several methods to set the state and get a bet, this seems excessive. For this simple case, the strategy can be a callable interface with a few public attributes.

The following is the straight betting strategy, which is always the same:

```
class BettingStrategy:
    def __init__(self) -> None:
       self.win = 0
       self.loss = 0
    def __call__(self) -> int:
        return 1
bet = BettingStrategy()
```

The idea of this interface is that a `Player` object will inform the betting strategy of win amounts and loss amounts. The `Player` object might have methods such as the following to inform the betting strategy about the outcome:

```
    def win(self, amount) -> None:
        self.bet.win += 1
        self.stake += amount
    def loss(self, amount) -> None:
         self.bet.loss += 1
         self.stake -= amount
```

These methods inform a betting strategy object (the `self.bet` object) whether the hand was a win or a loss. When it's time to place a bet, the `Player` object will perform something like the following operation to get the current betting level:

```
    def initial_bet(self) -> int:
        return self.bet()
```

This is a pleasantly short method implementation. After all, the betting strategy doesn't do much other than encapsulate a few, relatively simple rules.

The compactness of the callable interface can be helpful. We don't have many method names, and we don't have a complex set of syntaxes for a class to represent something as simple as bet amount.

Complexities and the callable interface

Let's see how well this interface design holds up as our processing becomes more complex. The following is the double-up on each loss strategy (also known as the **Martingale** betting system):

```
class BettingMartingale(BettingStrategy):

    def __init__(self) -> None:
        self._win = 0
        self._loss = 0
        self.stage = 1

    @property
    def win(self) -> int:
        return self._win

    @win.setter
    def win(self, value: int) -> None:
        self._win = value
        self.stage = 1

    @property
    def loss(self) -> int:
        return self._loss

    @loss.setter
    def loss(self, value: int) -> None:
        self._loss = value
        self.stage *= 2

    def __call__(self) -> int:
        return self.stage
```

Each loss doubles the betting by multiplying the stage by two. This goes on until we win and recoup our losses, reach the table limit, or go broke and can no longer place any bets. Casinos limit this by imposing table limits.

Whenever we win, the betting is reset to the base bet. The `stage` variable is reset to have a value of `1`.

The goal is to easily access an attribute value. The client of this class will be able to use `bet.win += 1`. This can depend on the property `setter` methods to make additional state changes based on the wins and losses. We only really care about the `setter` properties, but we must define the `getter` properties in order to clearly

create the `setter` properties. In addition to counting wins and losses, the `setter` methods also set the `stage` instance variable.

We can see this class in action in the following code snippet:

```
>>> bet= BettingMartingale()
>>> bet()
1
>>> bet.win += 1
>>> bet()
1
>>> bet.loss += 1
>>> bet()
2
```

The interface to this object is still quite simple. We can either count the wins and reset the bet to the base, or we can count the losses, and the bets will double.

The use of properties made the class definition long and hideous. Since we're really only interested in the `setter` properties and not the `getter` properties, we can use `__setattr__()` to streamline the class definition somewhat, as shown in the following code:

```
class BettingMartingale2(BettingStrategy):

    def __init__(self) -> None:
        self.win = 0
        self.loss = 0
        self.stage = 1

    def __setattr__(self, name: str, value: int) -> None:
        if name == "win":
            self.stage = 1
        elif name == "loss":
            self.stage *= 2
        super().__setattr__(name, value)

    def __call__(self) -> int:
        return self.stage
```

We used `__setattr__()` to monitor the changes to the `win` and `loss` attributes. In addition to setting the instance variables using `super().__setattr__()`, we also updated the internal state for the betting amount.

This is a nicer looking class definition, and it retains the same, simple interface as the original callable object with two attributes.

Managing contexts and the with statement

Contexts and context managers are used in several places in Python. We'll look at a few examples to establish the basic terminology.

Python defines context using the `with` statement. The following program is a small example that parses a log file to create a useful CSV summary of that log. Since there are two open files, this will use the nested `with` contexts. The example uses a complex regular expression, `format_1_pat`. We'll define this shortly.

We might see something similar to the following in an application program:

```
from pathlib import Path
import gzip
import csv

source_path = Path.cwd()/"data"/"compressed_data.gz"
target_path = Path.cwd()/"data"/"subset.csv"

with target_path.open('w', newline='') as target:
    wtr= csv.writer( target )
    with gzip.open(source_path) as source:
        line_iter = (b.decode() for b in source)
        row_iter = Counter(format_1_pat.match(line) for line in line_iter)
        non_empty_rows: Iterator[Match] = filter(None, row_iter)
        wtr.writerows(m.groups() for m in non_empty_rows)
```

Two contexts with two context managers are part of this example:

- The outermost context starts with the `with target_path.open('w', newline='') as target:` statement. The `path.open()` method opens a file that is also a context manager and assigns it to the `target` variable for further use.
- The inner context starts with the `with gzip.open(source_path, "r") as source:` statement. This `gzip.open()` function opens the given path and also behaves as a context manager.

When the `with` statements end, the contexts exit and the files are properly closed; this means that all of the buffers are flushed and the operating system resources are released. Even if there's an exception in the body of the `with` context, the context manager's exit will be processed correctly and the file will be closed.

Always use a with statement around a path.open() and related file-system operations

*Since files involve **operating system** (**OS**) resources, it's important to be sure that the entanglements between our applications and the OS are released as soon as they're no longer needed. The `with` statement ensures that resources are used properly.*

Just to complete the example, the following is the regular expression used to parse the Apache HTTP server log files in the **Common Log Format**:

```
import re
format_1_pat= re.compile(
    r"([\d\.]+)\s+" # digits and .'s: host
    r"(\S+)\s+"     # non-space: logname
    r"(\S+)\s+"     # non-space: user
    r"\[(.+?)\]\s+" # Everything in []: time
    r'"(.+?)"\s+'   # Everything in "": request
    r"(\d+)\s+"     # digits: status
    r"(\S+)\s+"     # non-space: bytes
    r'"(.*?)"\s+'   # Everything in "": referrer
    r'"(.*?)"\s*'   # Everything in "": user agent
)
```

The preceding expression located the various log format fields used in the previous example.

Using the decimal context

Another context that is used frequently is the decimal context. This context defines a number of properties of the `decimal.Decimal` calculation, including the quantization rules used to round or truncate values.

We might see application programming that looks similar to the following code snippet:

```
import decimal
PENNY = decimal.Decimal("0.00")

price = decimal.Decimal('15.99')
rate = decimal.Decimal('0.0075')
print(f"Tax={(price * rate).quantize(PENNY)}, Fully={price * rate}")

with decimal.localcontext() as ctx:
    ctx.rounding = decimal.ROUND_DOWN
    tax = (price*rate).quantize(PENNY)
print(f"Tax={tax}")
```

The preceding example shows both a default context **as well as** a local context. The default context is shown first, and it uses the default rounding rule.

The localized context begins with the `with decimal.localcontext() as ctx:` statement. Within this context, the decimal rounding has been defined to round down for this particular calculation.

The `with` statement is used to assure that the original context is restored after the localized change. Outside this context, the default rounding applies. Inside this context, a modified rounding rule applies.

Other contexts

There are a few other common contexts. Almost all of them are associated with basic input/output operations. Most modules that open a file create a context along with the file-like object.

Contexts are also associated with locking and database transactions. We may acquire and release an external lock, such as a semaphore, or we may want a database transaction to properly commit when it's successful or roll back when it fails. These are all the things that have defined contexts in Python.

The PEP 343 document provides a number of other examples of how the `with` statement and context managers might be used. There are also other places where we might like to use a context manager.

We may need to create classes that are simply context managers, or we may need to create classes that can have multiple purposes, one of which is to be a context manager. We'll look at a number of design strategies for contexts.

We'll return to this again in `Chapter 9`, *Decorators and Mixins – Cross-Cutting Aspects*, where we can cover a few more ways to create classes that have context manager features.

Defining the __enter__() and __exit__() methods

The defining feature of a context manager is that it has two special methods: `__enter__()` and `__exit__()`. These are used by the `with` statement to enter and exit the context. We'll use a simple context so that we can see how they work.

We'll often use context managers to make global state changes. This might be a change to the database transaction status or a change to the locking status of a resource, something that we want to do and then undo when the transaction is complete.

For this example, we'll make a global change to the random number generator. We'll create a context in which the random number generator uses a fixed and known seed, providing a fixed sequence of values.

The following is the context manager class definition:

```
import random
from typing import Optional, Type
from types import TracebackType

class KnownSequence:

    def __init__(self, seed: int = 0) -> None:
        self.seed = 0

    def __enter__(self) -> 'KnownSequence':
        self.was = random.getstate()
        random.seed(self.seed, version=1)
        return self

    def __exit__(
            self,
            exc_type: Optional[Type[BaseException]],
            exc_value: Optional[BaseException],
            traceback: Optional[TracebackType]
    ) -> Optional[bool]:
        random.setstate(self.was)
        return False
```

We defined the required `__enter__()` and `__exit__()` methods for the context manager. The `__enter__()` method will save the previous state of the random

module and then reset the seed to a given value. The `__exit__()` method will restore the original state of the random number generator.

Note that `__enter__()` returns `self`. This is common for **mixin** context managers that have been added into other class definitions. We'll look at the concept of a mixin in `Chapter 9`, *Decorators And Mixins – Cross-Cutting Aspects*. Note that the `__enter__()` method cannot have a type hint that refers to the `KnownSequence` class, because the class definition isn't complete. Instead, a string, `'KnownSequence'`, is used; mypy will resolve this to the class when the type hint checking is done.

The `__exit__()` method's parameters will have the value of `None` under normal circumstances. Unless we have specific exception-handling needs, we generally ignore the argument values. We'll look at exception handling in the following code. Here's an example of using the context to print five bunches of random numbers:

```
print(tuple(random.randint(-1,36) for i in range(5)))
with KnownSequence():
    print(tuple(random.randint(-1,36) for i in range(5)))
print(tuple(random.randint(-1,36) for i in range(5)))
with KnownSequence():
    print(tuple(random.randint(-1,36) for i in range(5)))
print(tuple(random.randint(-1,36) for i in range(5)))
```

The two groups of random numbers created within the context managed by an instance of `KnownSequence`, produce a fixed sequence of values. Outside the two contexts, the random seed is restored, and we get random values.

The output will look like the following (in most cases):

```
(12, 0, 8, 21, 6)
(23, 25, 1, 15, 31)
(6, 36, 1, 34, 8)
(23, 25, 1, 15, 31)
(9, 7, 13, 22, 29)
```

Some of this output is machine-dependent. While the exact values may vary, the second and fourth lines will match because the seed was fixed by the context. The other lines will not necessarily match, because they rely on the `random` module's own randomization features.

Handling exceptions

Exceptions that arise in a context manager's block will be passed to the `__exit__()` method of the context manager. The standard bits of an exception – the class, arguments, and the traceback stack – will all be provided as argument values.

The `__exit__()` method can do one of the following two things with the exception information:

- Silence the exception by returning some `True` value.
- Allow the exception to rise normally by returning any other `False` value. Returning nothing is the same as returning `None`, which is a `False` value; this allows the exception to propagate.

An exception might also be used to alter what the context manager does on exit. We might, for example, have to carry out special processing for certain types of OS errors that might arise.

Context manager as a factory

We can create a context manager class, which is a factory for an application object. This gives us a pleasant separation of design considerations without cluttering up an application class with context management features.

Let's say we want a deterministic `Deck` for dealing in Blackjack. This isn't as useful as it might sound. For unit testing, we'll need a complete mock deck with specific sequences of cards. This has the advantage that the context manager works with the classes we already saw.

We'll extend the simple context manager shown earlier to create a `Deck` that can be used within the `with` statement context.

The following is a class that is a factory for `Deck` and also tweaks the `random` module:

```
class Deterministic_Deck:

    def __init__(self, *args, **kw) -> None:
        self.args = args
        self.kw = kw

    def __enter__(self) -> Deck:
        self.was = random.getstate()
        random.seed(0, version=1)
        return Deck(*self.args, **self.kw)

    def __exit__(
            self,
            exc_type: Optional[Type[BaseException]],
            exc_value: Optional[BaseException],
            traceback: Optional[TracebackType]
    ) -> Optional[bool]:
        random.setstate(self.was)
        return False
```

The preceding context manager class preserves the argument values so that it can create a `Deck` with the given arguments.

The `__enter__()` method preserves the old random number state and then sets the `random` module in a mode that provides a fixed sequence of values. This is used to build and shuffle the deck.

Note that the `__enter__()` method returns a newly minted `Deck` object to be used in the `with` statement context. This is assigned via the `as` clause in the `with` statement. The type hint specifies `Deck` as the return type of this method. The following is a way to use this factory context manager:

```
with Deterministic_Deck(size=6) as deck:
    h = Hand(deck.pop(), deck.pop(), deck.pop())
```

The preceding example of code guarantees a specific sequence of cards that we can use for demonstration and testing purposes.

Cleaning up in a context manager

In this section, we'll discuss a more complex context manager that attempts some cleanup when there are problems.

This addresses the common issue where we want to save a backup copy of a file that our application is rewriting. We want to be able to do something similar to the following:

```
with Updating(some_path):
    with some_path.open('w') as target_file:
        process(target_file)
```

The intent is to have the original file renamed to `some_file copy`. If the context works normally, that is, no exceptions are raised, then the backup copy can be deleted or renamed to `some_file old`.

If the context doesn't work normally, that is, an exception is raised, we want to rename the new file to `some_file error` and rename the old file to `some_file`, putting the original file back the way it was before the exception.

We will need a context manager similar to the following:

```
from pathlib import Path
from typing import Optional

class Updating:

    def __init__(self, target: Path) -> None:
        self.target: Path = target
        self.previous: Optional[Path] = None

    def __enter__(self) -> None:
        try:
            self.previous = (
                self.target.parent
                    / (self.target.stem + " backup")
                ).with_suffix(self.target.suffix)
            self.target.rename(self.previous)
        except FileNotFoundError:
            self.previous = None

    def __exit__(
            self,
            exc_type: Optional[Type[BaseException]],
            exc_value: Optional[BaseException],
            traceback: Optional[TracebackType]
```

```
    ) -> Optional[bool]:
        if exc_type is not None:
            try:
                self.failure = (
                    self.target.parent
                        / (self.target.stem + " error")
                    ).with_suffix(self.target.suffix)
                self.target.rename(self.failure)
            except FileNotFoundError:
                pass  # Never even got created.
            if self.previous:
                self.previous.rename(self.target)
        return False
```

This context manager's `__enter__()` method will attempt to preserve any previous copy of the named file if it already exists. If it didn't exist, there's nothing to preserve. The file is preserved by simply renaming it to a name such as `"file backup.ext"`.

The `__exit__()` method will be given information about any exception that occurred in the body of context. If there is no exception, nothing more needs to be done. If there is an exception, then the `__exit__()` method will try to preserve the output (with a suffix of `"error"`) for debugging purposes. It will also put any previous copy of the file back in place by renaming the backup to the original name.

This is functionally equivalent to a `try-except-finally` block. However, it has the advantage that it separates the relevant application processing from the context management. The application processing is written in the `with` statement. The context issues are set aside into a separate class.

Summary

We looked at three of the special methods for class definition. The `__call__()` method is used when creating a callable. The callable is used to create functions that are stateful. Our primary example is a function that memoizes previous results.

The `__enter__()` and `__exit__()` methods are used to create a context manager. The context is used to handle processing that is localized to the body of a `with` statement. Most of our examples include input-output processing. Python also uses localized contexts for the decimal state. Other examples include making patches for unit testing purposes or acquiring and releasing locks.

Callable design considerations and trade-offs

When designing a callable object, we need to consider the following:

- The first consideration is the interface of the object. If there's a reason for the object to have a function-like interface, then a callable object is a sensible design approach. Using `collections.abc.Callable` assures that the callable API is built correctly, and it informs anyone reading the code what the intent of the class is.
- The secondary consideration is the statefulness of the function. Ordinary functions in Python have no hysteresis – there's no saved state. A callable object, however, can easily save a state. The memoization design pattern makes good use of stateful callable objects.

The only disadvantage of a callable object is the amount of syntax that is required. An ordinary function definition is shorter and therefore less error-prone and easier to read.

It's easy to migrate a defined function to a callable object, as follows:

```
def x(args):
    body
```

The preceding function can be converted into the following callable object:

```
class X:
    def __call__(self, args):
        body
x= X()
```

This is the minimal set of changes required to get the function to pass unit tests in the new form. The existing body will work in the new context unmodified.

Once the change has been made, features can be added to the callable object's version of the function.

Context manager design considerations and trade-offs

A context is generally used for acquire/release, open/close, and lock/unlock types of operation pairs. Most of the examples are file I/O related, and most of the file-like objects in Python are already proper context managers.

A context manager is almost always required for anything that has steps that bracket the essential processing. In particular, anything that requires a final `close()` method should be wrapped by a context manager.

Some Python libraries have open/close operations, but the objects aren't proper contexts. The `shelve` module, for example, doesn't create a proper context.

We can (and should) use the `contextllib.closing()` context on a `shelve` file. We'll show this in `Chapter 10`, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML*.

For our own classes that require a `close()` method, we can use the `closing()` function. When confronted with a class that has any kind of acquire/release life cycle, we want to acquire resources in `__init__()` or a class-level `open()` method and release them in `close()`. That way, our class can integrate well with this `closing()` function.

The following is an example of a class being wrapped that requires a `close()` function:

```
with contextlib.closing(MyClass()) as my_object:
    process(my_object)
```

The `contextllib.closing()` function will invoke the `close()` method of the object that is given as an argument. We can guarantee that `my_object` will have its `close()` method evaluated.

Looking forward

In the next two chapters, we'll look at the special methods used to create containers and numbers. In `Chapter 7`, *Creating Containers and Collections*, we'll look at the containers and collections in the standard library. We'll also look at building a unique, new kind of container. In `Chapter 8`, *Creating Numbers*, we'll look at the various numeric types and how we can create our own kind of number.

Creating Containers and Collections

We can extend a number of the standard library **abstract base classes** (**ABCs**) to create new kinds of collections. The ABCs also provide us with design guidelines to extend the built-in containers. These allow us to fine-tune the features or define new data structures that fit our problem domain more precisely.

We'll look at the basics of ABCs for container classes. There are a fairly large number of abstractions that are used to assemble the built-in Python types, such as `list`, `tuple`, `dict`, `set`, and `frozenset`. We'll review the variety of special methods that are involved in being a container and offering the various features of containers. We'll split these into the core container methods, separate from the more specialized `sequence`, `map`, and `set` methods. We'll address extending built-in containers in order to add features. We'll also look at wrapping built-in containers and delegating methods through the wrapper to the underlying container.

Finally, we'll look at building entirely new containers. This is a challenging territory, because there's a huge variety of interesting and useful collection algorithms already present in the Python Standard Library. In order to avoid deep computer science research, we'll build a pretty lame collection. Before starting on a real application, a careful study of *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein is essential. We'll finish by summarizing some of the design considerations that go into extending or creating new collections.

In this chapter, we will cover the following topics:

- ABCs of collections
- Examples of special methods
- Using the standard library extensions
- Creating new kinds of collections
- Narrowing a collection's type
- Defining a new kind of sequence
- Creating a new kind of mapping
- Creating a new kind of set

Technical requirements

The code files for this chapter are available at `https://git.io/fj2U2`.

ABCs of collections

The `collections.abc` module provides a wealth of ABCs that decompose collections into a number of discrete feature sets. A related set of features of a class is called a *protocol*: the idea is that things such as getting, setting, and deleting items are the protocol for list-like behavior. Similarly, the `__iter__()` method is part of the protocol for defining an iterable collection. A list often implements both protocols, but some data structures may support fewer protocols. Support for a given protocol is often exploited by **mypy** algorithms to determine whether an object is being used properly.

We can successfully use the `list` class without thinking too deeply about the various features and how they relate to the `set` class or the `dict` class. Once we start looking at the ABCs, however, we can see that there's a bit of subtlety to these classes. By decomposing the aspects of each collection, we can see areas of overlap that manifest themselves as an elegant polymorphism, even among different data structures.

At the bottom of the base classes are some definitions of the core protocols for collections.

These are the base classes that often define a single special method:

- The `Container` base class requires the concrete class to implement the `__contains__()` method. This special method implements the `in` operator.
- The `Iterable` base class requires `__iter__()`. This special method is used by the `for` statement and the generator expressions as well as the `iter()` function.
- The `Sized` base class requires `__len__()`. This method is used by the `len()` function. It's also prudent to implement `__bool__()`, but it's not required by this ABC.
- The `Hashable` base class requires `__hash__()`. This is used by the `hash()` function. If this is implemented, it means that the object is immutable.

Each of these abstract class definitions is used to build the higher-level, composite definitions of structures we can use in our applications. These composite constructs include the lower-level base classes of `Sized`, `Iterable`, and

`Container`. Here are some composite base classes that we might use in an application:

- The `Sequence` and `MutableSequence` classes build on the basics and include methods such as `index()`, `count()`, `reverse()`, `extend()`, and `remove()`.
- The `Mapping` and `MutableMapping` classes include methods such as `keys()`, `items()`, `values()`, and `get()`, among others.
- The `Set` and `MutableSet` classes include comparison and arithmetic operators to perform set operations.

If we look more deeply into the built-in collections, we can see how the ABC definitions serve to organize the special methods that we need to write or modify.

The `collections` module also contains three concrete implementations: `UserDict`, `UserList` and `UserString`. `UserDict` is a version of the built-in dictionary, with the details exposed. Similarly, `UserList` and `UserString` provide implementations that can be extended through subclasses. These can be helpful to see how a collection is built. In older versions of Python, these were used as superclasses and were extended because the built-in types could not easily be extended. In Python 3, the built-in types are trivially extended: these are rarely used except as example code.

Let's take a look at some examples of special methods in the next section.

Examples of special methods

When looking at a blackjack `Hand` object, we have an interesting special case for containment. We often want to know if there's an ace in the hand. If we define `Hand` as an extension of `list`, then we can't ask for a generic ace. We can only ask for specific cards. We have to write something like the following example:

```
any(c.rank == 'A' for c in hand.cards)
```

This examines each card serially. For a small collection where the checking is rare, the design has few consequences. If, on the other hand, we simulated millions of hands, this search would be repeated often enough that the cost would be troubling.

For other problem domains, where the collection may contain millions of items, we certainly can't scan millions of items serially. A better scheme for a collection of objects can be helpful. Ideally, we'd like something like this:

```
'A' in hand.cards
```

This means that we're modifying the meaning of *contains* for a `Hand` object that extends `list`. We're not looking for a `Card` instance; we're merely looking for the `rank` property of a `Card` object. We can override the `__contains__()` method to do this:

```
def __contains__(self, rank: int) -> bool:
    return any(c.rank==rank for rank in hand.cards)
```

This allows us to use a simpler `in` test for a given rank in a hand. The serial examination of individual cards is still present, but it's encapsulated within the `Hand` class, and we can introduce special-purpose indexes based on dictionaries to optimize this. Similar design considerations can be applied to the `__iter__()` and `__len__()`, special methods. Be cautious, however. Changing the semantics of `len()` or how a collection interacts with the `for` statement, might be disastrous.

The next section explains how to use the standard library extensions.

Using the standard library extensions

We'll look at some extensions to built-in classes that are already part of the standard library. These are the collections that extend or modify the built-in collections. Most of these are covered in one form or another in books such as *Python 3 Object-Oriented Programming - Third Edition* by Dusty Phillips.

We'll look at the following four collection from this library:

- `deque` (note the atypical class name) is a double-ended queue, a list-like collection that can perform fast appends and pops on either end. A subset of the features of this class will create single-ended stacks or queues.
- `ChainMap` is a view of multiple mappings. Instead of merging mappings together, we can keep them separate and *chain* among them to locate which mapping contains a requested key.
- `defaultdict` (note the atypical spelling) is a `dict` subclass that uses a factory function to provide values for missing keys.
- `Counter` is a `dict` subclass that can be used for counting objects to create frequency tables. However, it's actually a more sophisticated data structure called a **multiset** or **bag**.

There are two collections in this library that have been replaced by more advanced versions:

- The `namedtuple()` function creates a subclass of `tuple` with named attributes. This has been replaced by the `NamedTuple` definition in the `typing` module. We'll emphasize the new `typing.NamedTuple` class because it permits type hints. The legacy function is no longer useful.
- An `OrderedDict` collection is a mapping in which the original key entry order is maintained. This feature of maintaining key order is now a first-class part of the built-in `dict` class, so this special collection isn't necessary anymore.

We'll see examples of the preceding collection classes. There are two important lessons to be learned from studying the library collections:

- What features are already present the standard library; this will save us from reinvention

* How to extend the ABCs to add interesting and useful structures to the language

Also, it can be helpful to read the source for the libraries. The source will show us numerous Python object-oriented programming techniques. Beyond these basics are even more modules. They are as follows:

* The `heapq` module is a set of functions that impose a heap queue structure on an existing `list` object. The heap queue invariant is the set of those items in the heap that are maintained, in order to allow rapid retrieval in ascending order. If we use the `heapq` methods on a `list` structure, we will never have to explicitly sort the list. This can have significant performance improvements.
* The `array` module is a kind of sequence that optimizes storage for certain kinds of values. This provides list-like features over potentially large collections of simple values.

We won't provide detailed examples of these advanced modules. In addition, of course, there's the deeper computer science that supports these various data structure definitions.

Let's take a look at the different classes in the next few sections.

The typing.NamedTuple class

The `NamedTuple` class expects a number of class-level attributes. These attributes will typically have type hints, and provide a way to give names to the attributes of a tuple.

Using a `NamedTuple` subclass can condense a class definition into a very short definition of a simple immutable object. It saves us from having to write longer and more complex class definitions for the common case where we want to name a fixed set of attributes.

For something like a playing card, we might want to insert the following code in a class definition:

```
from typing import NamedTuple

class BlackjackCard_T(NamedTuple):
    rank: str
    suit: Suit
    hard: int
    soft: int
```

We defined a new class and provided four named attributes: `rank`, `suit`, `hard`, and `soft`. Since each of these objects is immutable, we don't need to worry about a badly behaved application attempting to change the rank of a `BlackjackCard` instance.

We can use a factory function to create instances of this class, as shown in the following code:

```
def card_t(rank: int, suit: Suit) -> BlackjackCard_T:
    if rank == 1:
        return BlackjackCard_T("A", suit, 1, 11)
    elif 2 <= rank < 11:
        return BlackjackCard_T(str(rank), suit, rank, rank)
    elif rank == 11:
        return BlackjackCard_T("J", suit, 10, 10)
    elif rank == 12:
        return BlackjackCard_T("Q", suit, 10, 10)
    elif rank == 13:
        return BlackjackCard_T("K", suit, 10, 10)
    else:
        raise ValueError(f"Invalid Rank {rank}")
```

The `card_t()` function will build an instance of `BlackjackCard` with the hard and soft totals set properly for various card ranks. The intent here is to use `card_t(7, Suit.Hearts)` to create an instance of the `BlackjackCard` class. The various points will be set automatically by the `card_t()` function.

A subclass of `NamedTuple` will include a class-level attribute named `_fields`, which names the fields. Additionally, a `_field_types` attribute provides details of the type hints provided for the attributes. These permit sophisticated introspection on `NamedTuple` subclasses.

We can, of course, include methods in a `NamedTuple` class definition. An example of including methods is as follows:

```
class BlackjackCard_T(NamedTuple):
    rank: str
    suit: Suit
    hard: int
    soft: int
    def is_ace(self) -> bool:
        return False

class AceCard(BlackjackCard):
    def is_ace(self) -> bool:
        return True
```

A subclass can't add any new attributes. The subclass can, however, meaningfully override method definitions. This technique can create usefully polymorphic subclasses of a `NamedTuple` class.

The deque class

A `list` object is designed to provide uniform performance for any element within the container. Some operations have performance penalties. Most notably, any operation extending from the front of a list, such as `list.insert(0, item)`, or removing from the front of a list, such as `list.pop(0)`, will incur some overheads because the list's size is changed, and the position of each element must then changed.

A `deque` – a double-ended queue – is designed to provide uniform performance for the first and last elements of a list. The idea is that appending and popping will be faster than the built-in `list` object.

> *Class names are usually in title case. However, the `deque` class doesn't follow the common pattern.*

Our design for a deck of cards avoids the potential performance pitfall of a `list` object by always popping from the end, never from the beginning. Using the default `pop()`, or an explicit `pop(-1)`, leverages the asymmetry of a list by using the low-cost location for removing at item.

The `deque.pop()` method is very fast and works from either end of the list. While this can be handy, we can check whether the performance of shuffling may suffer. A shuffle will make random access to the container, something for which `deque` is not designed.

In order to confirm the potential costs, we can use `timeit` to compare `list` and `deque` shuffling performance as follows:

```
>>> timeit.timeit('random.shuffle(x)',"""
... import random
... x=list(range(6*52))""")
597.951664149994
>>>
>>> timeit.timeit('random.shuffle(d)',"""
... from collections import deque
```

```
... import random
... d=deque(range(6*52))""")
609.9636979339994
```

We invoked `timeit` using `random.shuffle()`. The first example works on a `list` object; the second example works on a `deque` object.

These results indicate that shuffling a `deque` object is only a trifle slower than shuffling a `list` object – about 2 percent slower. This distinction is a hair not worth splitting. We can confidently try a `deque` object in place of `list`.

The change amounts to this:

```
from collections import dequeue
class Deck(dequeue):
    def __init__( self, size=1 ):
        super().__init__()
        for d in range(size):
           cards = [ card(r,s) for r in range(13) for s in Suits ]
            super().extend( cards )
        random.shuffle( self )
```

We replaced `list` with `deque` in the definition of `Deck`. Otherwise, the class is identical.

What is the actual performance difference? Let's create decks of 100,000 cards and deal them:

```
>>> timeit.timeit('x.pop()', "x=list(range(100000))",
 number=100000)
0.032304395994287916
>>> timeit.timeit('x.pop()', "from collections import deque;
 x=deque(range(100000))", number=100000)
0.013504189992090687
```

We invoked `timeit` using `x.pop()`. The first example works on a `list`; the second example works on a `deque` object.

The dealing time is cut almost by half (42 percent, actually). We had big savings from a tiny change in the data structure.

In general, it's important to pick the optimal data structure for the application. Trying several variations can show us what's more efficient.

The ChainMap use case

The use case for chaining maps together fits nicely with Python's concept of local versus global definitions. When we use a variable in Python, first the local namespaces and then the global namespaces are searched, in that order. In addition to searching both namespaces for a variable, setting a variable is done in the local namespace without disturbing the global namespace. This default behavior (without the `global` or `nonlocal` statements) is also how `ChainMap` works.

When our applications start running, we often have properties that come from command-line parameters, configuration files, OS environment variables, and possibly a default file of settings installed with the software. These have a clear precedence order where a value supplied on the command-line is most important, and an installation-wide default setting is the least important. Because a `ChainMap` object will search through the various mappings in order, it lets us merge many sources of parameters into a single dictionary-like structure, so that we can easily locate a setting.

We might have an application startup that combines several sources of configuration options like this:

```
import argparse
import json
import os
import sys
from collections import ChainMap
from typing import Dict, Any

def get_options(argv: List[str] = sys.argv[1:]) -> ChainMap:
    parser = argparse.ArgumentParser(
        description="Process some integers.")
    parser.add_argument(
        "-c", "--configuration", type=open, nargs="?")
    parser.add_argument(
        "-p", "--playerclass", type=str, nargs="?",
        default="Simple")
    cmdline = parser.parse_args(argv)

    if cmdline.configuration:
        config_file = json.load(cmdline.configuration)
        cmdline.configuration.close()
    else:
        config_file = {}

    default_path = (
        Path.cwd() / "Chapter_7" / "ch07_defaults.json")
```

```
with default_path.open() as default_file:
    defaults = json.load(default_file)

combined = ChainMap(
    vars(cmdline), config_file, os.environ, defaults)
return combined
```

The preceding code shows us the configuration from several sources, such as the following:

- The command-line arguments. In this example, there is only one argument, called `playerclass`, but a practical application will often have many, many more.
- One of the arguments, `configuration`, is the name of a configuration file with additional parameters. This is expected to be in the JSON format, and the file's contents are read.
- Additionally, there's a `defaults.json` file with yet another place to look for the configuration values.

From the preceding sources, we can build a single `ChainMap` object. This object permits looking for a parameter in each of the listed locations in the specified order. The `ChainMap` instance use case will search through each mapping from highest precedence to lowest, looking for the given key and returning the value. This gives us a tidy, easy-to-use source for runtime options and parameters.

We'll look at this again in `Chapter 14`, *Configuration Files and Persistence*, and `Chapter 18`, *Coping with the Command Line*.

The OrderedDict collection

The `OrderedDict` collection is a Python dictionary with an added feature. The order in which keys were inserted is retained.

One common use for `OrderedDict` is when processing HTML or XML files, where the order of objects must be retained, but objects might have cross-references via the `ID` and `IDREF` attributes. We can optimize the connections among objects by using the ID as a dictionary key. We can retain the source document's ordering with the `OrderedDict` structure.

With release 3.7, the built-in `dict` class makes the same guarantee of preserving the order in which dictionary keys were inserted. Here's an example:

```
>>> some_dict = {'zzz': 1, 'aaa': 2}
>>> some_dict['mmm'] = 3
>>> some_dict
{'zzz': 1, 'aaa': 2, 'mmm': 3}
```

In earlier releases of Python, the order of keys in a dictionary were not guaranteed to match the order in which they were inserted. The ordering of keys used to be arbitrary and difficult to predict. The `OrderedDict` class added the insertion order guarantee in these older releases of Python. Since the order of the keys is now guaranteed to be the order in which keys were inserted, the `OrderedDict` class is redundant.

The defaultdict subclass

An ordinary `dict` type raises an exception when a key is not found. A `defaultdict` collection class does this differently. Instead of raising an exception, it evaluates a given function and inserts the value of that function into the dictionary as a default value.

Class names are usually in upper TitleCase. However, the `defaultdict` class doesn't follow this pattern.

A common use case for the `defaultdict` class is to create indices for objects. When several objects have a common key, we can create a list of objects that share this key.

Here's a function that accumulates a list of distinct values based on a summary of the two values:

```
from typing import Dict, List, Tuple, DefaultDict
def dice_examples(n: int=12, seed: Any=None) -> DefaultDict[int, List]:
    if seed:
        random.seed(seed)
    Roll = Tuple[int, int]
    outcomes: DefaultDict[int, List[Roll]] = defaultdict(list)
    for _ in range(n):
        d1, d2 = random.randint(1, 6), random.randint(1, 6)
        outcomes[d1+d2].append((d1, d2))
    return outcomes
```

The type hint for `Roll` shows that we consider a roll of the dice to be a two-tuple composed of integers. The `outcomes` object has a hint that it will be a dictionary that has integer keys and the associated value will be a list of `Roll` instances.

The dictionary is built using `outcomes[d1+d2].append((d1, d2))`. Given two random numbers, `d1` and `d2`, the sum is the key value. If this key value does not already exist in the `outcomes` mapping, the `list()` function is used to build a default value of an empty list. If the key already exists, the value is simply fetched, and the `append()` method is used to accumulate the actual pair of numbers.

As another example, we can use the a `defaultdict` collection class to provide a constant value. We can use this instead of the `container.get(key,"N/A")` expression.

We can create a zero-argument `lambda` object. This works very nicely. Here's an example:

```
>>> from collections import defaultdict
>>> messages = defaultdict(lambda: "N/A")
>>> messages['error1']= 'Full Error Text'
>>> messages['other']
'N/A'
>>> messages['error1']
'Full Error Text'
```

In the first use of `messages['error1']`, a value was assigned to the `'error1'` key. This new value will replace the default. The second use of `messages['other']` will add the default value to the dictionary.

We can determine how many new keys were created by looking for all the keys that have a value of `"N/A"`:

```
>>> [k for k in messages if messages[k] == "N/A"]
['other']
```

As you can see in the preceding output, we found the key that was assigned the default value of `"N/A"`. This is often a helpful summary of the data that is being accumulated. It shows us all of the keys associated with the default value.

The counter collection

One of the most common use cases for a `defaultdict` class is when accumulating counts of key instances. A simple way to count keys looks like this:

```
frequency = defaultdict(int)
for k in some_iterator():
    frequency[k] += 1
```

This example counts the number of times each key value, `k`, appears in the sequence of values from `some_iterator()`.

This use case is so common that there's a variation on the `defaultdict` theme that performs the same operation shown in the preceding code–it's called `Counter`. A `Counter` collection, however, is considerably more sophisticated than a simple `defaultdict` class.

Here's an example that creates a frequency histogram from some source of data showing values in descending order by frequency:

```
from collections import Counter
frequency = Counter(some_iterator())
for k, freq in frequency.most_common():
    print(k, freq)
```

This example shows us how we can easily gather statistical data by providing any iterable item to `Counter`. It will gather frequency data on the values in that iterable item. In this case, we provided an iterable function named `some_iterator()`. We might have provided a sequence or some other collection.

We can then display the results in descending order of popularity. But wait! That's not all.

The `Counter` collection is not merely a simplistic variation of the `defaultdict` collection. The name is misleading. A `Counter` object is actually a aaaaaaaaaaaaaaaa*multiset*, sometimes called a **bag**.

It's a collection that is set-like, but allows repeat values in the bag. It is not a sequence with items identified by an index or position; order doesn't matter. It is

not a mapping with keys and values. It is like a set in which items stand for themselves and order doesn't matter. But, it is unlike a set because, in this case, elements can repeat.

As elements can repeat, the `Counter` object represents multiple occurrences with an integer count. Hence, it's used as a frequency table. However, it does more than this. As a bag is like a set, we can compare the elements of two bags to create a union or an intersection.

Let's create two bags:

```
>>> bag1 = Counter("aardwolves")
>>> bag2 = Counter("zymologies")
>>> bag1
Counter({'a': 2, 'o': 1, 'l': 1, 'w': 1, 'v': 1, 'e': 1,
'd': 1,  's': 1, 'r': 1})
>>> bag2
Counter({'o': 2, 'm': 1, 'l': 1, 'z': 1, 'y': 1, 'g': 1,
'i': 1,  'e': 1, 's': 1})
```

We built each bag by examining a sequence of letters. For characters that occur more than once, there's a count that is more than one.

We can easily compute the union of the two bags:

```
>>> bag1+bag2
Counter({'o': 3, 's': 2, 'l': 2, 'e': 2, 'a': 2, 'z': 1,
'y': 1,  'w': 1, 'v': 1, 'r': 1, 'm': 1, 'i': 1, 'g': 1,
'd': 1})
```

This shows us the entire suite of letters between the two strings. There were three instances of `o`. Not surprisingly, other letters were less popular.

We can just as easily compute the difference between the bags:

```
>>> bag1-bag2
Counter({'a': 2, 'w': 1, 'v': 1, 'd': 1, 'r': 1})
>>> bag2-bag1
Counter({'o': 1, 'm': 1, 'z': 1, 'y': 1, 'g': 1, 'i': 1})
```

The first expression shows us characters in `bag1` that were not in `bag2`.

The second expression shows us characters in `bag2` that were not in `bag1`. Note that the letter `o` occurred twice in `bag2` and once in `bag1`. The difference only removed one of the `o` characters from `bag1`.

In the next section, we'll see how to create new kinds of collections.

Creating new kinds of collections

We'll look at some extensions we might make to Python's built-in container classes. We won't show an example of extending each container.

We'll pick an example of extending a specific container and see how the process works:

1. Define the requirements. This may include research on Wikipedia, generally starting here: `http://en.wikipedia.org/wiki/Data_structure`. Designs of data structures often involve complex edge cases around missing items and duplicate items.
2. If necessary, look at the `collections.abc` module to see what methods must be implemented to create the new functionality.
3. Create some test cases. This also requires careful study of the algorithms to ensure that the edge cases are properly covered.
4. Write code based on the previous research steps.

We need to emphasize the importance of researching the fundamentals before trying to invent a new kind of data structure. In addition to searching the web for overviews and summaries, details will be necessary. See any of the following:

- *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein
- *Data Structures and Algorithms* by Aho, Ullman, and Hopcroft
- *The Algorithm Design Manual* by Steven Skiena

As we saw earlier, the ABCs define three broad kinds of collections: sequences, mappings, and sets. We have three design strategies that we can use to create new kinds of collections of our own:

- **Extend**: This is an existing sequence.
- **Wrap**: This is an existing sequence.
- **Invent**: This is a new sequence created from scratch.

In principle, we could give as many as nine examples – each basic flavor of collection with each basic design strategy. We won't beat this subject to death like that. We'll dig deep to create new kinds of sequences, learning how to

extend and wrap
existing sequences.

As there are so many extended mappings (such as `ChainMap`, `OrderedDict`, `defaultdict`, and `Counter`), we'll only touch lightly on creating new kinds of mappings. We'll also dig deep to create a new kind of ordered multiset or bag.

Let's narrow a collection's type in the next section.

Narrowing a collection's type

Python 3 allows us to provide extensive type hints for describing the contents of a collection. This has two benefits:

- It helps us visualize the data structure.
- It supports running **mypy** to confirm that the code uses the data structures properly.

The non-collection types (`int`, `str`, `float`, `complex`, and so on) all use the type name as their type hint. The built-in collections all have parallel type definitions in the `typing` module. It's common to see `from typing import List, Tuple, Dict, Set` to import these type names into a module.

Each of the type hints accepts parameters to further narrow the definition:

- The `List[T]` hint claims the object will be a `list` and all the items will be of type `T`. For example `[1, 1, 2, 3, 5, 8]` can be described as `List[int]`.
- A `Set[T]` hint is similar to the `List[T]` hint. It claims all items in the set will be of type `T`. For example, `{'a', 'r', 'd'}` can be described as `Set[str]`.
- The `Dict[K, V]` hint claims the object will be a `dict`, the keys will all have a type `K`, and the values will all have a type `V`. For example, `{'A': 4, 'B' 12}` can be described as `Dict[str, int]`.

The `Tuple` hint is often more complex. There are two common cases for tuples:

- A hint such as `Tuple[str, int, int, int]` describes a four-tuple with a string and three integer values, for example, `('crimson', 220, 20, 60)`. The size is specified explicitly.
- A hint such as `Tuple[int, ...]` describes a tuple of indefinite size. The items will all be type `int`. The size is not specified. The `...` notation is a token in the Python language, and is a first-class part of the syntax for this type hint.

To describe objects where `None` values may be present in a collection, the `Optional` type is used. We might have a type hint such as `List[Optional[int]]` to describe a list that is a mixture of `int` and `None` objects, for example `[1, 2, None, 42]`.

Because of the type coercion rules for numbers, we can often summarize numerical algorithms using a `float` type hint, such as the following:

```
def mean(data: List[float]) -> float: ...
```

This function will also work with a list of integer values. The **mypy** program recognizes the type coercion rules and will recognize `mean([8, 9, 10])` as a valid use of this function.

In the next section, we'll define a new kind of sequence.

Defining a new kind of sequence

A common requirement that we have when performing statistical analysis is to compute basic means, modes, and standard deviations on a collection of data. Our blackjack simulation will produce outcomes that must be analyzed statistically to see if we have actually invented a better strategy.

When we simulate the playing strategy for a game, we will develop some outcome data that will be a sequence of numbers that show the final result of playing the game several times with a given strategy.

We could accumulate the outcomes into a built-in `list` class. We can compute the mean via $\frac{\sum_{a \in x} a}{N}$, where N is the number of elements in x:

```
def mean(outcomes: List[float]) -> float:
    return sum(outcomes) / len(outcomes)
```

Standard deviation can be computed via $\frac{\sqrt{N \times \sum_{a \in x} a^2 - \left(\sum_{a \in x} a \right)^2}}{N}$:

```
def stdev(outcomes: List[float]) -> float:
    n = float(len(outcomes))
    return math.sqrt(
        n * sum(x**2 for x in outcomes) - sum(outcomes)**2
    ) / n
```

Both of these are relatively simple calculation functions. As things get more complex, however, loose functions such as these become less helpful. One of the benefits of object-oriented programming is to bind the functionality with the data.

Our first example will not involve rewriting any of the special methods of `list`. We'll simply subclass `list` to add methods that will compute the statistics. This is a very common kind of extension.

We'll revisit this in the second example so that we can revise and extend the

special methods. This will require some study of the ABC special methods to see what we need to add or modify so that our new list subclass properly inherits all the features of the built-in `list` class.

Because we're looking at sequences, we also have to wrestle with the Python `slice` notation. We'll look at what a slice is and how it works internally in the *Working with* `__getitem__`*,* `__setitem__`*,* `__delitem__`*, and slices* section.

The second important design strategy is wrapping. We'll create a wrapper around a list and see how we can delegate methods to the wrapped list. Wrapping has some advantages when it comes to object persistence, which is the subject of `Chapter 10`, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML*.

We can also look at the kinds of things that need to be done to invent a new kind of sequence from scratch.

A statistical list

It makes good sense to incorporate mean and standard deviation features directly into a subclass of `list`. We can extend `list` like this:

```
class StatsList(list):

    def __init__(self, iterable: Optional[Iterable[float]]) -> None:
        super().__init__(cast(Iterable[Any], iterable))

    @property
    def mean(self) -> float:
        return sum(self) / len(self)

    @property
    def stdev(self) -> float:
        n = len(self)
        return math.sqrt(
            n * sum(x ** 2 for x in self) - sum(self) ** 2
        ) / n
```

With this simple extension to the built-in `list` class, we can accumulate data and report statistics on the collection of data items.

Note the relative complexity involved in narrowing the type of the list class. The built-in list structure's type, `List`, is effectively `List[Any]`. In order for the arithmetic operations to work, the content really must be `List[float]`. By stating the `__init__()` method will only accept an `Iterable[float]` value, **mypy** is forced to confirm arguments to `StatsList` will meet this criteria. Let's imagine we have a source of raw data:

```
def data_gen() -> int:
    return random.randint(1, 6) + random.randint(1, 6)
```

This little `data_gen()` function is a stand-in for a variety of possible functions. It might be a complex simulation. It might be a source of actual measurements. The essential feature of this function—defined by the type hint—is to create an integer value.

We can imagine that an overall simulation script can use the `StatsList` class, as follows:

```
random.seed(42)
data = [data_gen() for _ in range(100)]
```

```
stats = StatsList(data)
print(f"mean = {stats.mean:f}")
print(f"stdev= {stats.stdev:.3f}")
```

This snippet uses a list comprehension to create a raw `list` object with 100 samples. Because the data object is built from the `data_gen()` function, it's clear that the data object has the type `List[int]`. From this, a `StatsList` object is created. The resulting `stats` object has `mean` and `stdev` properties, which are extensions to the base list class.

Choosing eager versus lazy calculation

Note that the calculations in the previous example are lazy; they are only done when requested. This also means that they're performed each and every time they're requested. This can be a considerable overhead, depending on the context in which objects of these classes are used.

It's may be sensible to transform these statistical summaries into eager calculations, as we know when elements are added and removed from a list. Although there's a hair more programming to create eager versions of these functions, it has a net impact of improving performance when there's a lot of data being accumulated.

The point of eager statistical calculations is to avoid the loops that compute sums. If we compute the sums eagerly, as the list is being created, we avoid extra looping through the data.

When we look at the special methods for a `Sequence` class, we can see all of the places where data is added to, removed from, and modified in the sequence. We can use this information to recompute the two sums that are involved. We start with the `collections.abc` section of the *Python Standard Library* documentation, section 8.4.1, at http://docs.python.org/3.4/library/collections.abc.html#collections-abstract-base-classes.

Here are the required methods for a `MutableSequence` class: `__getitem__`, `__setitem__`, `__delitem__`, `__len__`, `insert`, `append`, `reverse`, `extend`, `pop`, `remove`, and `__iadd__`. The documentation also mentions the *Inherited Sequence methods*. However, as they are for immutable sequences, we can certainly ignore them.

Here are the details of what must be done to update the statistical results in each method:

- `__getitem__`: There's no change in the state.
- `__setitem__`: This changes an item. We need to take the old item out of each

sum and fold the new item into each sum.

- `__delitem__`: This removes an item. We need to take the old item out of each sum.
- `__len__`: There's no change in the state.
- `insert`: This adds a new item. We need to fold it into each sum.
- `append`: This also adds a new item. We need to fold it into each sum.
- `reverse`: There's no change in the value of the mean or standard deviation.
- `extend`: This adds many new items. All of the items must be folded into the sums.
- `pop`: This removes an item. We need to take the old item out of each sum.
- `remove`: This removes an item. We need to take the old item out of each sum.
- `__iadd__`: This is the `+=` augmented assignment statement, the in-place addition. It's effectively the same as the `extend` keyword.

We won't look at each method in detail, because there are really combinations of two use cases:

- Fold in one new value
- Remove one old value

The replacement case is a combination of the remove and fold in operations.

Here are the elements of an eager `StatsList2` class. We're going to see the `insert()` and `pop()` method:

```
class StatsList2(list):
    """Eager Stats."""

    def __init__(self, iterable: Optional[Iterable[float]]) -> None:
        self.sum0 = 0  # len(self), sometimes called "N"
        self.sum1 = 0.0  # sum(self)
        self.sum2 = 0.0  # sum(x**2 for x in self)
        super().__init__(cast(Iterable[Any], iterable))
        for x in self:
            self._new(x)

    def _new(self, value: float) -> None:
        self.sum0 += 1
        self.sum1 += value
        self.sum2 += value * value

    def _rmv(self, value: float) -> None:
        self.sum0 -= 1
        self.sum1 -= value
        self.sum2 -= value * value

    def insert(self, index: int, value: float) -> None:
        super().insert(index, value)
```

```
        self._new(value)

    def pop(self, index: int = 0) -> None:
        value = super().pop(index)
        self._rmv(value)
        return value
```

We provided three internal variables with comments to show the invariants that this class will maintain. We'll call these the *sum invariants* because each of them contains a particular kind of sum that is maintained as invariant (always true) after each kind of state change. The essence of this eager calculation are the `_rmv()` and `_new()` methods, which update our three internal sums based on changes to the list, so that the relationships really remain invariant.

When we remove an item, that is, after a successful `pop()` operation, we have to adjust our sums. When we add an item (either initially, or via the `insert()` method), we also have to adjust our sums. The other methods we need to implement will make use of these two methods to ensure that the three sum invariants hold. For a given list of values, L, we ensure that `L.sum0` is always $\sum_{x \in L} x^0 = \sum_{x \in L} 1$, `L.sum1` is always $\sum_{x \in L} x$, and `L.sum2` is always $\sum_{x \in L} x^2$. We can use the sums to compute the mean and standard deviation.

Other methods, such as `append()`, `extend()`, and `remove()`, are similar in many ways to these methods. We haven't shown them because they're similar.

We can see how this list works by playing with some data:

```
>>> sl2 = StatsList2( [2, 4, 3, 4, 5, 5, 7, 9, 10] )
>>> sl2.sum0, sl2.sum1, sl2.sum2
(9, 49, 325)
>>> sl2[2]= 4
>>> sl2.sum0, sl2.sum1, sl2.sum2
(9, 50, 332)
>>> del sl2[-1]
>>> sl2.sum0, sl2.sum1, sl2.sum2
(8, 40, 232)
>>> sl2.insert( 0, -1 )
>>> sl2.pop()
-1
>>> sl2.sum0, sl2.sum1, sl2.sum2
(8, 40, 232)
```

We can create a list and the sums are computed initially. Each subsequent change eagerly updates the various sums. We can change, remove, insert, and pop an item; each change results in a new set of sums.

All that's left, is to add our mean and standard deviation calculations, which we can do as follows:

```
@property
def mean(self) -> float:
    return self.sum1 / self.sum0

@property
def stdev(self) -> float:
    return math.sqrt(
        self.sum0*self.sum2 - self.sum1*self.sum1
    ) / self.sum0
```

These make use of the sums that have already been computed. There's no additional looping over the data to compute these two statistics.

Working with __getitem__(), __setitem__(), __delitem__(), and slices

The `StatsList2` example didn't show us the implementation of `__setitem__()` or `__delitem__()` because they involve slices. We'll need to look at the implementation of a slice before we can implement these methods properly.

Sequences have two different kinds of indexes:

- `a[i]`. This is a simple integer index.
- `a[i:j]` or `a[i:j:k]`: These are `slice` expressions with `start:stop:step` values. Slice expressions can be quite complex, with seven different variations for different kinds of defaults.

This basic syntax works in three contexts:

- In an expression, relying on `__getitem__()` to get a value
- On the left-hand side of assignment, relying on `__setitem__()` to set a value
- On a `del` statement, relying on `__delitem__()` to delete a value

When we do something like `seq[:-1]`, we write a `slice` expression. The underlying `__getitem__()` method will be given a `slice` object, instead of a simple integer.

The reference manual tells us a few things about slices. A `slice` object will have three attributes: `start`, `stop`, and `step`. It will also have a method function called `indices()`, which will properly compute any omitted attribute values for a slice.

We can explore the `slice` objects with a trivial class that extends `list`:

```
class Explore(list):
    def __getitem__( self, index ):
        print( index, index.indices(len(self)) )
        return super().__getitem__( index )
```

This class will dump the `slice` object and the value of the `indices()` function result.

Then, use the superclass implementation, so that the list behaves normally otherwise.

Given this class, we can try different `slice` expressions to see what we get:

```
>>> x = Explore('abcdefg')
>>> x[:]
slice(None, None, None) (0, 7, 1)
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> x[:-1]
slice(None, -1, None) (0, 6, 1)
['a', 'b', 'c', 'd', 'e', 'f']
>>> x[1:]
slice(1, None, None) (1, 7, 1)
['b', 'c', 'd', 'e', 'f', 'g']
>>> x[::2]
slice(None, None, 2) (0, 7, 2)
['a', 'c', 'e', 'g']
```

In the preceding `slice` expressions, we can see that a `slice` object has three attributes, and the values for those attributes come directly from the Python syntax. When we provide the proper length to the `indices()` function, it returns a three-tuple value with start, stop, and step values.

Implementing __getitem__(), __setitem__(), and __delitem__()

When we implement the `__getitem__()`, `__setitem__()` and `__delitem__()` methods, we must work with two kinds of argument values: `int` and `slice`. This variant behavior requires two different type hints. The hints are provided using the `@overload` decorator.

When we overload the various sequence methods, we must handle the slice situation appropriately within the body of the method. This requires using the `isinstance()` function to discern whether a `slice` object or a simple `int` has been provided as an argument value.

Here is a `__setitem__()` method that works with slices:

```
@overload
def __setitem__(self, index: int, value: float) -> None: ...

@overload
def __setitem__(self, index: slice, value: Iterable[float]) -> None: ...

def __setitem__(self, index, value) -> None:
    if isinstance(index, slice):
        start, stop, step = index.indices(len(self))
        olds = [self[i] for i in range(start, stop, step)]
        super().__setitem__(index, value)
        for x in olds:
            self._rmv(x)
        for x in value:
            self._new(x)
    else:
        old = self[index]
        super().__setitem__(index, value)
        self._rmv(old)
```

The preceding method has two processing paths:

- If the index is a `slice` object, we'll compute the `start`, `stop`, and `step` values. Then, we'll locate all the old values that will be removed. We can then invoke the superclass operation and fold in the new values that replaced the old values.
- If the index is a simple `int` object, the old value is a single item, and the new value is also a single item.

Note that `__setitem__()` expected multiple type hints, written using the `@overload` descriptor. The `__delitem__()` definition, on the other hand, relies on `Union[int, slice]`, instead of two overloaded definitions.

Here's the `__delitem__()` method, which works with either a slice or an integer:

```
def __delitem__(self, index: Union[int, slice]) -> None:
    # Index may be a single integer, or a slice
    if isinstance(index, slice):
        start, stop, step = index.indices(len(self))
        olds = [self[i] for i in range(start, stop, step)]
        super().__delitem__(index)
        for x in olds:
            self._rmv(x)
    else:
        old = self[index]
        super().__delitem__(index)
        self._rmv(old)
```

The preceding code, too, expands the slice to determine what values could be removed. If the index is a simple integer, then just one value is removed.

When we introduce proper slice processing to our `StatsList2` class, we can create lists that do everything the base `list` class does, and also (rapidly) return the mean and standard deviation for the values that are currently in the list.

Note that these method functions will each create a temporary list object, `olds`; this involves some overhead that can be removed. As an exercise for the reader, it's helpful to rewrite the `_rmv()` function to eliminate the use of the `olds` variable.

Wrapping a list and delegating

We'll look at how we might wrap one of Python's built-in container classes. Wrapping an existing class means that some methods will have to be delegated to the underlying container.

As there are a large number of methods in any of the built-in collections, wrapping a collection may require a fair amount of code. When it comes to creating persistent classes, wrapping has advantages over extending. That's the subject of `Chapter 10`, *Serializing and Saving - JSON, YAML, Pickle, CSV, and XML*. In some cases, we'll want to expose the internal collection to save writing a large number of sequence methods that delegate to an internal list.

A common restriction that applies to statistics data classes is that they need to be *insert only*. We'll be disabling a number of method functions. This is the kind of dramatic change in the class features that suggests using a wrapper class instead of an extension.

We can design a class that supports only `append` and `__getitem__`, for example. It would wrap a `list` class. The following code can be used to accumulate data from simulations:

```
class StatsList3:
    def __init__(self) -> None:
        self._list: List[float] = list()
        self.sum0 = 0  # len(self), sometimes called "N"
        self.sum1 = 0.  # sum(self)
        self.sum2 = 0.  # sum(x**2 for x in self)

    def append(self, value: float) -> None:
        self._list.append(value)
        self.sum0 += 1
        self.sum1 += value
        self.sum2 += value * value

    def __getitem__(self, index: int) -> float:
        return self._list.__getitem__(index)

    @property
    def mean(self) -> float:
        return self.sum1 / self.sum0

    @property
    def stdev(self) -> float:
        return math.sqrt(
```

```
            self.sum0*self.sum2 - self.sum1*self.sum1
        ) / self.sum0
```

This class has an internal `_list` object that is the underlying list. We've provided an explicit type hint to show the object is expected to be `List[float]`. The list is initially empty. As we've only defined `append()` as a way to update the list, we can maintain the various sums easily. We need to be careful to delegate the work to the superclass to be sure that the list is actually updated before our subclass processes the argument value.

We can directly delegate `__getitem__()` to the internal list object without examining the arguments or the results.

We can use this class as follows:

```
>>> sl3 = StatsList3()
>>> for data in 2, 4, 4, 4, 5, 5, 7, 9:
...     sl3.append(data)
...
>>> sl3.mean
5.0
>>> sl3.stdev
2.0
```

We created an empty list and appended items to the list. As we maintain the sums as items are appended, we can compute the mean and standard deviation extremely quickly.

We did not provide a definition of `__iter__()`. Instances of this class will, however, be iterable in spite of this omission.

Because we've defined `__getitem__()`, several things now work. Not only can we get items, but it also turns out that there will be a default implementation that allows us to iterate through the sequence of values.

Here's an example:

```
>>> sl3[0]
2
>>> for x in sl3:
...     print(x)
...
2
4
4
4
5
```

```
5
7
9
```

The preceding output shows us that a minimal wrapper around a collection is often enough to satisfy many use cases.

Note that we didn't, for example, make the list sizeable. If we attempt to get the size, it will raise an exception, as shown in the following snippet:

```
>>> len(sl3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'StatsList3' has no len()
```

We might want to add a `__len__()` method that delegates the real work to the internal `_list` object. We might also want to set `__hash__` to `None`, which would be prudent as this is a mutable object.

We might want to define `__contains__()` and delegate this feature to the internal `_list` too. This will create a minimalist container that offers the low-level feature set of a container.

Creating iterators with __iter__()

When our design involves wrapping an existing class, we'll need to be sure our class is iterable. When we look at the documentation for `collections.abc.Iterable`, we see that we only need to define `__iter__()` to make an object iterable. The `__iter__()` method can either return a proper `Iterator` object, or it can be a generator function.

Creating an `Iterator` object, while not terribly complex, is rarely necessary. It's much simpler to create generator functions. For a wrapped collection, we should always simply delegate the `__iter__()` method to the underlying collection.

For our `StatsList3` class, it would look like this:

```
    def __iter__(self):
        return iter(self._list)
```

This method function would delegate the iteration to the underlying list object's `Iterator`.

Creating a new kind of mapping

Python has a built-in mapping called `dict`, and numerous library mappings. In addition to the `collections` module extensions to `dict` (`defaultdict`, `Counter`, and `ChainMap`), there are several other library modules that contain mapping-like structures.

The `shelve` module is an important example of another mapping. We'll look at this in `Chapter 11`, *Storing and Retrieving Objects via Shelve*. The `dbm` module is similar to `shelve`, in that it also maps a key to a value.

The `mailbox` module and `email.message` modules both have classes that provide an interface that is similar to `dict` for the mailbox structure used to manage local e-mails.

As far as design strategies go, we can extend or wrap one of the existing mappings to add even more features.

We could upgrade `Counter` to add the mean and standard deviation to the data stored as a frequency distribution. Indeed, we can also calculate the median and mode very easily from this class.

Here's a `StatsCounter` extension to `Counter` that adds some statistical functions:

```
import math
from collections import Counter

class StatsCounter(Counter):
    @property
    def mean(self) -> float:
        sum0 = sum(v for k, v in self.items())
        sum1 = sum(k * v for k, v in self.items())
        return sum1 / sum0

    @property
    def stdev(self) -> float:
        sum0 = sum(v for k, v in self.items())
        sum1 = sum(k * v for k, v in self.items())
        sum2 = sum(k * k * v for k, v in self.items())
        return math.sqrt(sum0 * sum2 - sum1 * sum1) / sum0
```

We extended the `Counter` class with two new methods to compute the mean and standard deviation from the frequency distributions. The formulae are similar to

the examples shown earlier for the eager calculations on a `list` object, even though they're lazy calculations on a `Counter` object.

We used `sum0 = sum(v for k,v in self.items())` to compute a sum of the values, `v`, ignoring the `k` keys. We could use an underscore (`_`) instead of `k` to emphasize that we're ignoring the keys. We could also use `sum(v for v in self.values())` to emphasize that we're not using the keys. We prefer obvious parallel structures for `sum0` and `sum1`.

We can use this class to efficiently gather statistics and to perform quantitative analysis on the raw data. We might run a number of simulations, using a `Counter` object to gather the results.

Here's an interaction with a list of sample data that stands in for real results:

```
>>> sc = StatsCounter( [2, 4, 4, 4, 5, 5, 7, 9] )
>>> sc.mean
5.0
>>> sc.stdev
2.0
>>> sc.most_common(1)
[(4, 3)]
>>> list(sorted(sc.elements()))
[2, 4, 4, 4, 5, 5, 7, 9]
```

The results of `most_common()` are reported as a sequence of two-tuples with the mode value (4) and the number of times the value occurred (3). We might want to get the top three values to bracket the mode with the next two less-popular items. We get several popular values with an evaluation such as `sc.most_common(3)`.

The `elements()` method reconstructs a `list` that's like the original data with the items repeated properly.

From the sorted elements, we can extract the median, the middle-most item:

```
@property
def median(self) -> Any:
    all = list(sorted(self.elements()))
    return all[len(all) // 2]
```

This method is not only lazy, it's rather extravagant with memory; it creates an entire sequence of the available values merely to find the middle-most item.

While it is simple, this is often an expensive way to use Python.

A smarter approach would be to compute the effective length and mid-point via `sum(self.values())//2`. Once this is known, the keys can be visited in that order, using the counts to compute the range of positions for a given key. Eventually, a key will be found with a range that includes the midpoint.

The code would look something like the following:

```
@property
def median2(self) -> Optional[float]:
    mid = sum(self.values()) // 2
    low = 0
    for k, v in sorted(self.items()):
        if low <= mid < low + v: return k
        low += v
    return None
```

We stepped through the keys and the number of times they occur to locate the key that is midmost. Note that this uses the internal `sorted()` function, which is not without its own cost.

Via `timeit`, we can learn that the extravagant version takes 9.5 seconds; the smarter version takes 5.2 seconds.

Let's take a look at how to create a new kind of set in the next section.

Creating a new kind of set

Creating a whole new collection requires some preliminary work. We need to have new algorithms or new internal data structures that offer significant improvements over the built-in collections. It's important to do thorough *Big-O* complexity calculations before designing a new collection. It's also important to use `timeit` after an implementation to be sure that the new collection really is an improvement over available built-in classes.

We might, for example, want to create a binary search tree structure that will keep the elements in a proper order. As we want this to be a mutable structure, we'll have to perform the following kinds of design activities:

- Design the essential binary tree structure.
- Decide which structure is the basis: `MutableSequence`, `MutableMapping`, or `MutableSet`.
- Look at the special methods for the collection in the `collections.abc` section of the *Python Standard Library* documentation, section 8.4.1.

A binary search tree has nodes that contain a key value, and two branches: a *less than* branch for all keys less than this node's key, and a *greater than or equal to* branch for keys greater than, or equal to, this node's key.

We need to examine the fit between our collection and the Python ABCs:

- A binary tree doesn't fit well with some sequence features. Notably, we don't often use an integer index with a binary tree. We most often refer to elements in a search tree by their key. While we can impose an integer index without too much difficulty, it will involve $\mathbf{O}(n)$ tree traversals.
- A tree could be used for the keys of a mapping; this would keep the keys in a sorted order at a relatively low cost.
- It is a good alternative to a `set` or a `Counter` class because it can tolerate multiple copies of a key, making it easily bag-like.

We'll look at creating a sorted multiset or a bag. This can contain multiple copies of an object. It will rely on relatively simple comparison tests among objects.

This is a rather complex design. There are a great many details. To create a background, it's important to read articles such as `http://en.wikipedia.org/wiki/Binary_search_tree`. At the end of the previous Wikipedia page are a number of external links that will provide further information. It's essential to study the essential algorithms in books such as *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein; *Data Structures and Algorithms* by Aho, Ullman, and Hopcroft; or *The Algorithm Design Manual* by Steven Skiena.

Some design rationale

We're going to split the collection into two classes: `TreeNode` and `Tree`. This will allow us to separate the design into the essential data collection, and a Pythonic Façade design pattern required to match other collections.

The `TreeNode` class will contain the item as well as the `more`, `less`, and `parent` references. This is the core collection of values. It handles insertion and removal. Also, searching for a particular item in order to use `__contains__()` or `discard()` will be delegated to the `TreeNode` class.

The essential search algorithm's outline looks like this.

- If the target item is equal to the self item, then return `self`.
- If the target item is less than `self.item`, then recursively use `less.find(target item)`.
- If the target item is greater than `self.item`, then recursively use `more.find(target.item)`.

We'll use similar delegation to the `TreeNode` class for more of the real work of maintaining the tree structure.

We'll use the **Façade** design pattern to wrap details of the `TreeNode` with a Pythonic interface. This will define the visible, external definition the `Tree` itself. A Façade design can also be called a **wrapper**; the idea is to add features required for a particular interface. The `Tree` class provides the external interface required by a `MutableSet` ABC and keeps these requirements distinct from the implementation details in the `TreeNode` class.

The algorithms can be somewhat simpler if there's a root node that's empty and always compares less than all other key values. This can be challenging in Python because we don't know in advance what types of data the nodes might have; we can't easily define a bottom value for the root node. Instead, we'll use a special case value of `None`, and endure the overheads of `if` statements checking for the root node.

Defining the Tree class

We'll start with the wrapper or Façade class, `Tree`. This is the core of an extension to the `MutableSet` class that provides the minimal method functions:

```
class Tree(collections.abc.MutableSet):

    def __init__(self, source: Iterable[Comparable] = None) -> None:
        self.root = TreeNode(None)
        self.size = 0
        if source:
            for item in source:
                self.root.add(item)
                self.size += 1

    def add(self, item: Comparable) -> None:
        self.root.add(item)
        self.size += 1

    def discard(self, item: Comparable) -> None:
        if self.root.more:
            try:
                self.root.more.remove(item)
                self.size -= 1
            except KeyError:
                pass
        else:
            pass

    def __contains__(self, item: Any) -> bool:
        if self.root.more:
            self.root.more.find(cast(Comparable, item))
            return True
        else:
            return False

    def __iter__(self) -> Iterator[Comparable]:
        if self.root.more:
            for item in iter(self.root.more):
                yield item
        # Otherwise, the tree is empty.

    def __len__(self) -> int:
        return self.size
```

The initialization design is similar to that of a `Counter` object; this class will accept an iterable and load the elements into the structure. The source of data is provided with a type hint of `Iterable[Comparable]`. This hint imposes a restriction on the kinds of items which this collection can handle. If the collection is used with items that do not support the proper comparable protocol methods, then **mypy** will report an error.

Here's the definition of the `Comparable` type hint:

```
class Comparable(metaclass=ABCMeta):
    @abstractmethod
    def __lt__(self, other: Any) -> bool: ...
    @abstractmethod
    def __ge__(self, other: Any) -> bool: ...
```

The `Comparable` class definition is an abstraction which requires two methods: `__lt__()` and `__ge__()`. This is the minimum required for a class of objects to work properly with the `<`, `<=`, `>`, and `>=` operators. This defines the comparable protocol among objects that can be sorted or ranked.

The `add()` and `discard()` methods both update the tree, and also keep track of the overall size. That saves counting nodes via a recursive traversal of the tree. These methods also delegate their work to the `TreeNode` object at the root of the tree.

The `__contains__()` special method performs a recursive find. The initial check to be sure the tree contains a value in the root node is required by **mypy**. Without the `if` statement, the type hints suggest the `more` attribute could be `None`.

The `__iter__()` special method is a generator function. It also delegates the real work to recursive iterators within the `TreeNode` class.

We defined `discard()`; mutable sets require this to be silent when attempting to discard the missing keys. The abstract superclass provides a default implementation of `remove()`, which raises an exception if a key is not found. Both method functions must be present; we defined `discard()` based on `remove()`, by silencing the exception. In some cases, it might be easier to define `remove()` based on `discard()`, by raising an exception if a problem is found.

Because this class extends the `MutableSet` abstraction, numerous features are provided automatically. This saves us from copy-and-paste programming to create a number of boilerplate features. In some cases, our data structure may have more efficient implementations than the defaults, and we may want to override additional methods inherited from the abstract superclass.

Defining the TreeNode class

The overall `Tree` class relies on the `TreeNode` class to handle the implementation details of adding, removing, and iterating through the various items in the bag. This class is rather large, so we'll present it in four sections.

The first part shows the essential elements of initialization, representation, and how the attributes are made visible:

```
class TreeNode:

    def __init__(
        self,
        item: Optional[Comparable],
        less: Optional["TreeNode"] = None,
        more: Optional["TreeNode"] = None,
        parent: Optional["TreeNode"] = None,
    ) -> None:
        self.item = item
        self.less = less
        self.more = more
        if parent:
            self.parent = parent

    @property
    def parent(self) -> Optional["TreeNode"]:
        return self.parent_ref()

    @parent.setter
    def parent(self, value: "TreeNode"):
        self.parent_ref = weakref.ref(value)

    def __repr__(self) -> str:
        return f"TreeNode({self.item!r}, {self.less!r}, {self.more!r})"
```

Each individual node must have a reference to an item. Additional nodes with items that compare as less than the given item, or more than the given item are optional. Similarly, a parent node is also optional.

The property definitions for the `parent` methods are used to ensure that the parent attribute is actually a `weakref` attribute, but it appears like a strong reference. For more information on weak references, see `Chapter 3`, *Integrating Seamlessly - Basic Special Methods*. We have mutual references between a `TreeNode` parent object and its children objects; this circularity could make it difficult to remove `TreeNode` objects. Using a `weakref` breaks the circularity, allowing reference counting to remove nodes quickly when they are no longer referenced.

Note that the `TreeNode` type hints are references to the class from inside the class definition. This circularity can be a syntax problem because the class hasn't been fully defined. To make valid self-referential type hints, **mypy** allows the use of a string. When **mypy** is run, the string resolves to the proper type object.

Here are the methods for finding and iterating through the nodes:

```
def find(self, item: Comparable) -> "TreeNode":
    if self.item is None:  # Root
        if self.more:
            return self.more.find(item)
    elif self.item == item:
        return self
    elif self.item > item and self.less:
        return self.less.find(item)
    elif self.item < item and self.more:
        return self.more.find(item)
    raise KeyError

def __iter__(self) -> Iterator[Comparable]:
    if self.less:
        yield from self.less
    if self.item:
        yield self.item
    if self.more:
        yield from self.more
```

We saw the `find()` method, which performs a recursive search from a tree through the appropriate subtree looking for the target item. There are a total of six cases:

- When this is the root node of the tree, we'll simply skip it.
- When this node has the target item, we'll return it.
- When the target item is less than this node's item and there is a branch on the *less* side, we'll descend into that subtree to search.
- When the target item is more than this node's item and there is a branch on the *more* side, we'll descend into that subtree to search.
- There are two remaining cases: the target item is less than the current node, but there's no *less* branch or the target item is more than the current node, but there's no *more* branch. These two cases both mean the item cannot be found in the tree, leading to a `KeyError` exception.

The `__iter__()` method does what's called an inorder traversal of this node and its subtrees. As is typical, this is a generator function that yields the values from iterators over each collection of subtrees. Although we could create a separate iterator class that's tied to our `Tree` class, there's little benefit when this generator function does everything we need.

The result of the `__iter__()` has a type hint of `Iterator[Comparable]`. This reflects the minimal constraint placed on the items contained in the tree.

Here's the next part of this class to add a new node to a tree:

```
def add(self, item: Comparable):
    if self.item is None:  # Root Special Case
        if self.more:
            self.more.add(item)
        else:
            self.more = TreeNode(item, parent=self)
    elif self.item >= item:
        if self.less:
            self.less.add(item)
        else:
            self.less = TreeNode(item, parent=self)
    elif self.item < item:
        if self.more:
            self.more.add(item)
        else:
            self.more = TreeNode(item, parent=self)
```

This is the recursive search for the proper place to add a new node. The structure parallels the `find()` method.

Finally, we have the (more complex) processing to remove a node from the tree. This requires some care to relink the tree around the missing node:

```
def remove(self, item: Comparable):
    # Recursive search for node
    if self.item is None or item > self.item:
        if self.more:
            self.more.remove(item)
        else:
            raise KeyError
    elif item < self.item:
        if self.less:
            self.less.remove(item)
        else:
            raise KeyError
    else:  # self.item == item
        if self.less and self.more:  # Two children are present
            successor = self.more._least()
            self.item = successor.item
            if successor.item:
                successor.remove(successor.item)
        elif self.less:  # One child on less
            self._replace(self.less)
        elif self.more:  # One child on more
            self._replace(self.more)
        else:  # Zero children
            self._replace(None)

def _least(self) -> "TreeNode":
    if self.less is None:
        return self
    return self.less._least()
```

```
def _replace(self, new: Optional["TreeNode"] = None) -> None:
    if self.parent:
        if self == self.parent.less:
            self.parent.less = new
        else:
            self.parent.more = new
    if new is not None:
        new.parent = self.parent
```

The `remove()` method has two sections. The first part is the recursive search for the target node.

Once the node is found, there are three cases to consider:

- When we delete a node with no children, we simply delete it and update the parent to replace the link with `None`. The use of a weak reference from the removed node back to the parent, means memory cleanup and reuse is immediate.
- When we delete a node with one child, we can push the single child up to replace this node under the parent.
- When there are two children, we need to restructure the tree. We locate the successor node (the least item in the `more` subtree). We can replace the to-be-removed node with the content of this successor. Then, we can remove the duplicative former successor node.

We rely on two private methods. The `_least()` method performs a recursive search for the least-valued node in a given tree. The `_replace()` method examines a parent to see whether it should touch the `less` or `more` attribute.

Demonstrating the binary tree bag

We built a complete new collection. The ABC definitions included a number of methods automatically. These inherited methods might not be particularly efficient, but they're defined, they work, and we didn't write the code for them.

```
>>> s1 = Tree(["Item 1", "Another", "Middle"])
>>> list(s1)
['Another', 'Item 1', 'Middle']
>>> len(s1)
3
>>> s2 = Tree(["Another", "More", "Yet More"])
>>>
>>> union= s1 | s2
>>> list(union)
['Another', 'Another', 'Item 1', 'Middle', 'More', 'Yet More']
>>> len(union)
6
>>> union.remove('Another')
>>> list(union)
['Another', 'Item 1', 'Middle', 'More', 'Yet More']
```

This example shows us that the set `union` operator for set objects works properly, even though we didn't provide code for it specifically. As this is a bag, items are duplicated properly, too.

Design considerations and tradeoffs

When working with containers and collections, we have a multistep design strategy:

1. Consider the built-in versions of sequence, mapping, and set.
2. Consider the library extensions in the collection module, as well as extras such as `heapq`, `bisect`, and `array`.
3. Consider a composition of existing class definitions. In many cases, a list of `tuple` objects or a `dict` of lists provides the needed features.
4. Consider extending one of the earlier mentioned classes to provide additional methods or attributes.
5. Consider wrapping an existing structure as another way to provide additional methods or attributes.
6. Finally, consider a novel data structure. Generally, there is a lot of careful analysis available. Start with Wikipedia articles such as this one: `http://en.wikipedia.org/wiki/List_of_data_structures`.

Once the design alternatives have been identified, there are two parts of the evaluation left:

- How well the interface fits with the problem domain. This is a relatively subjective determination.
- How well the data structure performs as measured with `timeit`. This is an entirely objective result.

It's important to avoid the paralysis of analysis. We need to *effectively* find the proper collection.

In most cases, it is best to profile a working application to see which data structure is the performance bottleneck. In some cases, consideration of the complexity factors for a data structure will reveal its suitability for a particular kind of problem before starting the implementation.

TIP

Perhaps the most important consideration is this: for the highest performance, avoid search.

Avoiding search is the reason sets and mappings require hashable objects. A hashable object can be located in a set or mapping with almost no processing. Locating an item by value (not by index) in a list can take a great deal of time.

Here's a comparison of a bad set-like use of a list and a proper use of a set:

```
>>> import timeit
>>> timeit.timeit('l.remove(10); l.append(10)', 'l =
 list(range(20))')
0.8182099789992208
>>> timeit.timeit('l.remove(10); l.add(10)', 'l = set(range(20))')
0.30278149300283985
```

We removed and added an item from a list, as well as a set.

Clearly, abusing a list to get it to perform set-like operations makes the collection take 2.7 times as long to run.

As a second example, we'll abuse a list to make it mapping-like. This is based on a real-world example where the original code had two parallel lists to mimic the keys and values of a mapping.

We'll compare a proper mapping with two parallel lists, as follows:

```
>>> timeit.timeit('i = k.index(10); v[i]= 0', 'k=list(range(20));
 v=list(range(20))')
0.6549435159977293
>>> timeit.timeit('m[10] = 0', 'm=dict(zip(list(range(20)),list(range(20))))' )
0.0764331009995658
```

The first example uses two parallel lists. One list is used to look up a value, and then a change is made to the parallel list. In the other case, we simply updated a mapping.

Clearly, performing an index and update on two parallel lists is a horrifying mistake. It takes 8.6 times as long to locate something via `list.index()` as it does to locate it via a mapping and the hash value.

Summary

In this chapter, we looked at a number of built-in class definitions. The built-in collections are the starting place for most design work. We'll often start with `tuple`, `list`, `dict`, or `set`. We can leverage the extension to `tuple`, created by `namedtuple()` for an application's immutable objects.

Beyond these classes, we have other standard library classes in the `collections` mode that we can use:

- `deque`
- `ChainMap`
- `defaultdict`
- `Counter`

We have three standard design strategies, too. We can wrap any of these existing classes, or we can extend a class.

Finally, we can also invent an entirely new kind of collection. This requires defining a number of method names and special methods.

In the next chapter, we'll closely look at the built-in numbers and how to create new kinds of numbers. As with containers, Python offers a rich variety of built-in numbers. When creating a new kind of number, we'll have to define numerous special methods.

After looking at numbers, we can look at some more sophisticated design techniques. We'll look at how we can create our own decorators and use those to simplify the class definition. We'll also look at using mixin class definitions, which are similar to the ABC definitions.

Creating Numbers

We can extend the ABC abstractions in the `numbers` module to create new kinds of numbers. We might need to do this to create numeric types that fit our problem domain more precisely than the built-in numeric types. The abstractions in the `numbers` module need to be looked at first, because they define the existing built-in classes. Before working with new kinds of numbers, it's essential to see how the existing numbers work.

We'll digress to look at Python's operator-to-method mapping algorithm. The idea is that a binary operator has two operands; either operand can define the class that implements the operator. Python's rules for locating the relevant class are essential to decide what special methods to implement.

The essential arithmetic operators, such as `+`, `-`, `*`, `/`, `//`, `%`, and `**`, form the backbone of numeric operations. There are additional operators that include `^`, `|`, and `&`; these are used for the bit-wise processing of integers. They're also used as operators among sets. There are some more operators in this class, including `<<` and `>>`. The comparison operators were covered in `Chapter 3`, *Integrating Seamlessly – Basic Special Methods*. These include `<`, `>`, `<=`, `>=`, `==`, and `!=`. We'll review and extend our study of the comparison operators in this chapter.

There are a number of additional special methods for numbers. These include the various conversions to other built-in types. Python also defines *in-place* combinations of an assignment with an operator. These include `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `&=`, `|=`, `^=`, `>>=`, and `<<=`. These are more appropriate for mutable objects than numbers. We'll finish the chapter by summarizing some of the design considerations that go into extending or creating new numbers.

In this chapter, we will cover the following topics:

- ABCs of numbers
- The arithmetic operator's special method
- Creating a numeric class
- Computing a numeric hash
- Implementing other special methods

- Optimization with the in-place operators

Technical requirements

The code files for this chapter are available at `https://git.io/fj2Ua`.

ABCs of numbers

The `numbers` package provides a tower of numeric types that are all implementations of `numbers.Number`. Additionally, the `fractions` and `decimal` modules provide extension numeric types: `fractions.Fraction` and `decimal.Decimal`. These definitions roughly parallel the mathematical thought on the various classes of numbers. An article available at http://en.wikipedia.org/wiki/Number_theory contains numerous links to in-depth explanations; for example, *An Introduction to the Theory of Numbers*.

The essential question is how well computers can implement the underlying mathematical abstractions. To be more specific, we want to be sure that anything that is computable in the abstract world of mathematics can be computed (or approximated) using a concrete computer. This is why the question of computability is so important. The idea behind a *Turing complete* programming language is that it can compute anything that is computable by an abstract Turing machine. A helpful article can be found at http://en.wikipedia.org/wiki/Computability_theory; additional links in this article include https://www.encyclopediaofmath.org/index.php?title=p/t094460.

I recommend reading the Wikipedia articles on number theory and computability theory, and other concepts that I discuss in the chapter. The articles and the additional links present in the reference sections of these sources will give you more background information than can be covered here.

Python defines the following abstractions and their associated implementation classes. Further, these classes form an inheritance hierarchy, where each abstract class inherits from the class above it. As we move down the list, the classes have more features. As there are very few classes, as follows, it forms a *tower* rather than a tree:

- `numbers.Complex` implemented by `complex`
- `numbers.Real` implemented by `float`
- `numbers.Rational` implemented by `fractions.Fraction`
- `numbers.Integral` implemented by `int`

Additionally, we have `decimal.Decimal`, which is a bit like `float` ; it isn't a proper

subclass of `numbers.Real`, but is somewhat like it. While it may be obvious, it's still essential to repeat the fact that the `float` value is merely an approximation; it is not exact.

Don't be surprised by this sort of thing. The following example shows how a `float` value is only an approximation for real numbers:

```
>>> (105+(1/29)-105)*29
0.9999999999998153
```

Ordinary algebra suggests that this value should be equal to one. Because of the nature of the floating-point approximations, the actual result differs from the abstract ideal. In addition to the numeric class definitions, there are also a number of conversions among the various classes. It's not possible to convert from every type to every other type, so we must work out a matrix that shows the conversions that work and those conversions that can't work. The following list is a summary:

- `complex`: This can't be converted to any other type. A `complex` value can be decomposed into the `real` and `imag` portions, both of which are `float` values.
- `float`: This can be converted explicitly to any type, including `decimal.Decimal`. Arithmetic operators won't implicitly coerce a `float` value to `Decimal`.
- `Fractions.Fraction`: This can be converted to any of the other types, except `decimal.Decimal`. To get to `decimal` requires a two-part operation: first to `float`, and second to `decimal.Decimal`. This leads to an approximation.
- `int`: This can be converted to any of the other types.
- `Decimal`: This can be converted to any other type. It is not implicitly coerced to other types via arithmetic operations.

The up and down conversions come from the tower of numeric abstractions mentioned earlier.

Let's see how to decide which type to use in the next section.

Deciding which types to use

Because of the conversions, we see the following four general domains of numerical processing:

- **Complex:** Once we get involved in complex math, we'll be using `complex`, and `float`, plus the `cmath` module. We probably aren't going to use `Fraction` or `Decimal` at all. However, there's no reason to impose restrictions on the numeric types; most numbers can be converted to `complex`.
- **Currency:** For currency-related operations, we absolutely must use `Decimal`. Generally, when doing currency calculations, there's no good reason to mix the `decimal` values with non-decimal values. Sometimes, we'll use the `int` values, but there's no good reason to work with `float` or `complex` along with `Decimal`. Remember, floats are approximations, and that's unacceptable when working with currency.
- **Bit Kicking:** For operations that involve bit and byte processing, we'll generally use `int`, only `int`, and nothing but `int`.
- **Conventional:** This is a broad, vague *everything else* category. For most conventional mathematical operations, `int`, `float`, and `Fraction` are all interchangeable. Indeed, a well-written function can often be properly polymorphic; it will work perfectly well with any numeric type. Python types, particularly `float` and `int`, will participate in a variety of implicit conversions. This makes the selection of a specific numeric type for these kinds of problems somewhat moot.

These are generally obvious aspects of a given problem domain. It's often easy to distinguish applications that might involve science or engineering and complex numbers from applications that involve financial calculations, currency, and decimal numbers. It's important to be as permissive as possible in the numeric types that are used in an application. Needlessly narrowing the domain of types via the `isinstance()` test is often a waste of time and code.

In the next section, we'll talk about method resolution and the reflected operator concept.

Method resolution and the reflected operator concept

The arithmetic operators (`+`, `-`, `*`, `/`, `//`, `%`, `**`, and so on) all map to special method names. When we provide an expression such as `355+113`, the generic `+` operator will be mapped to a concrete `__add__()` method of a specific numeric class. This example will turn out to be evaluated as though we had written `355.__add__(113)`. The simplest rule is that the left-most operand determines the class of the operator being used.

But wait, there's more! When we have an expression with mixed types, Python may end up with two available implementations of the special method, one in each class. Consider `7-0.14` as an expression. Using the left-side `int` class, this expression will be attempted as `7.__sub__(0.14)`. This involves an unpleasant complexity, since the argument to an `int` operator is a `float` value `0.14` and converting `float` to `int` could potentially lose precision. Converting up the tower of types (from `int` toward `complex`) won't lose precision. Converting down the tower of types implies a potential loss of precision.

Using the right-hand side `float` version, however, this expression will be attempted as `0.14.__rsub__(7)`. In this case, the argument to a `float` operator is an `int` value `7`; converting `int` up the tower to `float` doesn't (generally) lose precision. (A truly giant `int` value can lose precision; however, that's a technical quibble, not a general principle.)

The `__rsub__()` operation is called **reflected subtraction**. The `X.__sub__(Y)` operation is the expected subtraction, and the `A.__rsub__(B)` operation is the reflected subtraction; the implementation method in the latter comes from the right-hand side operand's class. So far, we've seen the following two rules:

- Rule one: try the left-hand side operand's class first. If that works, good. If the operand returns `NotImplemented` as a value, then use rule two.
- Rule two: try the right-hand side operand with the reflected operator. If that works, good. If it returns `NotImplemented`, then it really is not implemented, so an exception must be raised.

The notable exception is when the two operands happen to have a subclass relationship.

The following additional rule applies before the first pair rules as a special case:

- If the right operand is a subclass of the left, and the subclass defines the reflected special method name for the operator, then the subclass reflected operator will be tried. This allows a subclass override to be used, even if the subclass operand is on the right-hand side of the operator.
- Otherwise, use rule one and try the left side.

Imagine we wrote a subclass of `float`, called `MyFloat`. In an expression such as `2.0-MyFloat(1)`, the right operand is of a subclass of the left operand's class. Because of this subclass relationship, `MyFloat(1).__rsub__(2.0)` will be tried first. The point of this rule is to give precedence to the subclass.

This means that a class that will do implicit coercion from other types must implement the forward as well as the reflected operators. When we implement or extend a numeric type, we must work out the conversions that our type is able to do.

In the next section, we'll take a look at the arithmetic operator's special methods.

The arithmetic operator's special methods

There are a total of 13 binary operators and their associated special methods. We'll focus on the obvious arithmetic operators first. The special method names match the operators (and functions), as shown in the following table:

Method	Operator
`object.__add__(self, other)`	`+`
`object.__sub__(self, other)`	`-`
`object.__mul__(self, other)`	`*`
`object.__truediv__(self, other)`	`/`
`object.__floordiv__(self, other)`	`//`
`object.__mod__(self, other)`	`%`
`object.__divmod__(self, other)`	`divmod()`
`object.__pow__(self, other[, modulo])`	`pow()` as well as `**`

And yes, interestingly, two functions are included with the various symbolic operators. There are a number of unary operators and functions, which have special method names as shown in the following table:

Method	Operator
`object.__neg__(self)`	`-`
`object.__pos__(self)`	`+`
`object.__abs__(self)`	`abs()`
`object.__complex__(self)`	`complex()`
`object.__int__(self)`	`int()`
`object.__float__(self)`	`float()`
`object.__round__(self[, n])`	`round()`
`object.__trunc__(self[, n])`	`math.trunc()`
`object.__ceil__(self[, n])`	`math.ceil()`
`object.__floor__(self[, n])`	`math.floor()`

And yes, there are a lot of functions in this list, too. We can tinker with Python's

internal trace to see what's going on under the hood. We'll define a simplistic trace function that will provide us with a little bit of visibility into what's going on:

```
def trace(frame, event, arg):
    if frame.f_code.co_name.startswith("__"):
        print(frame.f_code.co_name, frame.f_code.co_filename, event)
```

This function will dump special method names when the code associated with the traced frame has a name that starts with `"__"`. We can install this trace function in Python using the following code:

```
import sys
sys.settrace(trace)
```

Once installed, everything passes through our `trace()` function. We're filtering the trace events for special method names. We'll define a subclass of a built-in class so that we can explore the method resolution rules:

```
class noisyfloat( float ):
    def __add__( self, other ):
        print( self, "+", other )
        return super().__add__( other )
    def __radd__( self, other ):
        print( self, "r+", other )
        return super().__radd__( other )
```

This class overrides just two of the operator's special method names. When we add `noisyfloat` values, we'll see a printed summary of the operation. Plus, the trace will tell us what's going on. The following is the interaction that shows Python's choice of class to implement a given operation:

```
>>> x = noisyfloat(2)
>>> x+3
__add__ <stdin> call
2.0 + 3
5.0
>>> 2+x
__radd__ <stdin> call
2.0 r+ 2
4.0
>>> x+2.3
__add__ <stdin> call
2.0 + 2.3
4.3
>>> 2.3+x
__radd__ <stdin> call
2.0 r+ 2.3
4.3
```

From `x+3`, we see how `noisyfloat+int` provided the `int` object, `3`, to the `__add__()` method. This value was passed to the superclass, `float`, which handled the coercion of `3` to a `float` and did the addition, too. `2+x` shows how the right-hand side `noisyfloat` version of the operation was used. Again, `int` was passed to the superclass that handled the coercion to `float`. From `x+2.3`, we come to know that `noisyfloat+float` used the subclass that was on the left-hand side. On the other hand, `2.3+x` shows how `float+noisyfloat` used the subclass on the right-hand side and the reflected `__radd__()` operator.

Let's see how to create a numeric class.

Creating a numeric class

We'll try to design a new kind of number. This is no easy task when Python already offers integers of indefinite precision, rational fractions, standard floats, and decimal numbers for currency calculations. There aren't many features missing from this list.

We'll define a class of *scaled* numbers. These are numbers that include an integer value coupled with a scaling factor. We can use these for currency calculations. For many currencies of the world, we can use a scale of 100 and do all our calculations to the nearest cent.

The advantage of scaled arithmetic is that it can be done very simply by using low-level hardware instructions. We could rewrite this module to be a C-language module and exploit hardware speed operations. The disadvantage of inventing new scaled arithmetic is that the `decimal` package already does a very neat job of precise decimal arithmetic.

We'll call this `FixedPoint` class because it will implement a kind of fixed decimal point number. The scale factor will be a simple integer, usually a power of ten. In principle, a scaling factor that's a power of two could be considerably faster, but wouldn't be ideally suited for currency.

The reason a power of two scaling factor can be faster is that we can replace `value*(2**scale)` with `value << scale`, and replace `value/(2**scale)` with `value >> scale`. The left and right `shift` operations are often hardware instructions that are much faster than multiplication or division.

Ideally, the scaling factor is a power of ten, but we don't explicitly enforce this. It's a relatively simple extension to track both scaling power and the scale factor that goes with the power. We might store two as the power and as the factor. We've simplified this class definition to just track the factor.

Let's see how to define `FixedPoint` initialization in the next section.

Defining FixedPoint initialization

We'll start with initialization, which includes conversions of various types to the `FixedPoint` values as follows:

```
import numbers
import math
from typing import Union, Optional, Any

class FixedPoint(numbers.Rational):
    __slots__ = ("value", "scale", "default_format")

    def __init__(self, value: Union['FixedPoint', int, float], scale: int = 100) -> None
        self.value: int
        self.scale: int
        if isinstance(value, FixedPoint):
            self.value = value.value
            self.scale = value.scale
        elif isinstance(value, int):
            self.value = value
            self.scale = scale
        elif isinstance(value, float):
            self.value = int(scale * value + .5)  # Round half up
            self.scale = scale
        else:
            raise TypeError(f"Can't build FixedPoint from {value!r} of {type(value)}")
        digits = int(math.log10(scale))
        self.default_format = "{{0:.{digits}f}}".format(digits=digits)

    def __str__(self) -> str:
        return self.__format__(self.default_format)

    def __repr__(self) -> str:
        return f"{self.__class__.__name__:s}({self.value:d},scale={self.scale:d})"

    def __format__(self, specification: str) -> str:
        if specification == "":
            specification = self.default_format
        return specification.format(self.value / self.scale)
```

Our `FixedPoint` class is defined as a `numbers.Rational` subclass. We're going to wrap two integer values, `scale` and `value`, and follow the general definitions for fractions. This requires a large number of special method definitions. The initialization is for an immutable object, so it overrides `__new__()` instead of `__init__()`. It defines a limited number of slots to prevent the adding of any further attributes. The initialization includes several kinds of conversions as follows:

- If we're given another `FixedPoint` object, we'll copy the internal attributes to create a new `FixedPoint` object that's a kind of clone of the original. It will

have a unique ID, but we can be sure it has the same hash value and compares as equal, making the clone largely indistinguishable.
- When given integral or rational values (concrete classes of `int` or `float`), these are used to set the `value` and `scale` attributes.
- We can add cases to handle `decimal.Decimal` and `fractions.Fraction`, as well as parsing input string values.

We've defined three special methods to produce string results: `__str__()`, `__repr__()`, and `__format__()`. For the format operation, we've decided to leverage the existing floating-point features of the format specification language. Because this is a rational number, we need to provide numerator and denominator methods.

Note that we could have also started with wrapping the existing `fractions.Fraction` class. In order to show more of the programming required, we opted to start from the abstract `Rational` class.

Let's see how to define `FixedPoint` binary arithmetic operators in the next section.

Defining FixedPoint binary arithmetic operators

The sole reason for defining a new class of numbers is to overload the arithmetic operators. Each `FixedPoint` object has two parts: `value` and `scale`. We can say some value, A, is a fraction of the value A_v divided by the scaling factor A_s: $A = \frac{A_v}{A_s}$.

Note that we've worked out the algebra in the following example using correct, but inefficient, floating point expressions, we'll discuss the slightly more efficient, pure integer operations.

The general form for addition (and subtraction) is this: $A + B = \frac{A_v}{A_s} + \frac{B_v}{B_s} = \frac{A_v B_s + B_v A_s}{A_s B_s}$, which can create a result with a lot of useless precision.

Imagine adding 9.95 and 12.95. We'd have (in principle) 229000/10000. This can be properly reduced to 2290/100. The problem is that it also reduces to 229/10, which is no longer in cents. We'd like to avoid reducing fractions in a general way and instead stick with cents or mills to the extent possible.

We can identify two cases for $A + B = \frac{A_v}{A_s} + \frac{B_v}{B_s}$:

- **The scale factors match**: In this case, $A_s = B_s$ and the sum is $A + B = \frac{A_v}{A_s} + \frac{B_v}{A_s} = \frac{A_v + B_v}{A_s}$. When adding a `FixedPoint` value and a plain old integer value, this will also work. We can force the integer to have the required scale factor.

- **The scale factors don't match**: When $A_s \neq B_s$, it can help to produce a result scale R_s with the maximum scale factor of the two input values: $R_s = \max(A_s, B_s)$. From this, we can compute two intermediate scale factors: $\frac{R_s}{A_s}$ and $\frac{R_s}{B_s}$. One of those scale factors will be 1, and the other will

be less than 1. We can now add with a common value in the denominator. Algebraically, it's $\frac{\frac{A_v R_s}{A_s}}{\frac{A_s R_s}{A_s}} + \frac{\frac{B_v R_s}{B_s}}{\frac{B_s R_s}{B_s}} = \frac{\frac{A_v R_s}{A_s} + \frac{B_v R_s}{B_s}}{R_s}$. This can be further optimized into two cases, since one of the factors is 1 and the other is a power of 10.

We can't really optimize multiplication. It's $A \times B = \frac{A_v}{A_s} \times \frac{B_v}{B_s} = \frac{A_v B_v}{A_s B_s}$. The precision really must increase when we multiply the `FixedPoint` values.

Division is multiplication by an inverse: $A \div B = \frac{A_v}{A_s} \times \frac{B_s}{B_v} = \frac{A_v B_s}{A_s B_v}$. If A and B have the same scale, these values will cancel so that we do have a handy optimization available. However, this changes the scale from cents to wholes, which might not be appropriate.

The following is what the forward operators, built around a similar boilerplate, look like:

```
def __add__(self, other: Union['FixedPoint', int]) -> 'FixedPoint':
    if not isinstance(other, FixedPoint):
        new_scale = self.scale
        new_value = self.value + other * self.scale
    else:
        new_scale = max(self.scale, other.scale)
        new_value = self.value * (new_scale // self.scale) + other.value * (
            new_scale // other.scale
        )
    return FixedPoint(int(new_value), scale=new_scale)

def __sub__(self, other: Union['FixedPoint', int]) -> 'FixedPoint':
    if not isinstance(other, FixedPoint):
        new_scale = self.scale
        new_value = self.value - other * self.scale
    else:
        new_scale = max(self.scale, other.scale)
        new_value = self.value * (new_scale // self.scale) - other.value * (
            new_scale // other.scale
        )
    return FixedPoint(int(new_value), scale=new_scale)

def __mul__(self, other: Union['FixedPoint', int]) -> 'FixedPoint':
    if not isinstance(other, FixedPoint):
        new_scale = self.scale
        new_value = self.value * other
    else:
        new_scale = self.scale * other.scale
        new_value = self.value * other.value
    return FixedPoint(int(new_value), scale=new_scale)

def __truediv__(self, other: Union['FixedPoint', int]) -> 'FixedPoint':
```

```
        if not isinstance(other, FixedPoint):
            new_value = int(self.value / other)
        else:
            new_value = int(self.value / (other.value / other.scale))
        return FixedPoint(new_value, scale=self.scale)

    def __floordiv__(self, other: Union['FixedPoint', int]) -> 'FixedPoint':
        if not isinstance(other, FixedPoint):
            new_value = int(self.value // other)
        else:
            new_value = int(self.value // (other.value / other.scale))
        return FixedPoint(new_value, scale=self.scale)

    def __mod__(self, other: Union['FixedPoint', int]) -> 'FixedPoint':
        if not isinstance(other, FixedPoint):
            new_value = (self.value / self.scale) % other
        else:
            new_value = self.value % (other.value / other.scale)
        return FixedPoint(new_value, scale=self.scale)

    def __pow__(self, other: Union['FixedPoint', int]) -> 'FixedPoint':
        if not isinstance(other, FixedPoint):
            new_value = (self.value / self.scale) ** other
        else:
            new_value = (self.value / self.scale) ** (other.value / other.scale)
        return FixedPoint(int(new_value) * self.scale, scale=self.scale)
```

For the simple addition, subtraction, and multiplication cases, we've provided versions that can be optimized to eliminate some of the relatively slow floating point intermediate results.

Each of these operators returns an instance of the `FixedPoint` class. We can't use the name inside the class definition itself. We've provided a string version of the name. The **mypy** utility will resolve this string to the proper type name when it is used to check the type hints.

In some cases, we've used `Union['FixedPoint', int]` to explicitly support integer coercion. This type hint tells that **mypy** the method will accept either an instance of the `FixedPoint` class or a simple `int` object.

For the two divisions, the `__mod__()`, and `__pow__()` methods, we haven't done any optimization to try and eliminate noise being introduced via floating-point division. Instead, we've provided a working Python implementation that can be used with a suite of unit tests as a basis for optimization and refactoring.

It's important to note that the division operations can properly reduce the scale factors. However, changing the scale may be undesirable. When doing currency work, we might divide the currency rate (dollars) by a non-currency value (hours) to get the dollars-per-hour result. The proper answer might have zero

relevant decimal places. This would be a scale of 1, but we might want to force the value to have a cents-oriented scale of 100. This implementation ensures that the left-hand side operand dictates the desired number of decimal places.

Now, let's see how to define `FixedPoint` unary arithmetic operators.

Defining FixedPoint unary arithmetic operators

The following are the unary operator method functions:

```
def __abs__(self) -> 'FixedPoint':
    return FixedPoint(abs(self.value), self.scale)

def __float__(self) -> float:
    return self.value / self.scale

def __int__(self) -> int:
    return int(self.value / self.scale)

def __trunc__(self) -> int:
    return int(math.trunc(self.value / self.scale))

def __ceil__(self) -> int:
    return int(math.ceil(self.value / self.scale))

def __floor__(self) -> int:
    return int(math.floor(self.value / self.scale))

def __round__(self, ndigits: Optional[int] = 0) -> Any:
    return FixedPoint(round(self.value / self.scale, ndigits=ndigits), self.scale)

def __neg__(self) -> 'FixedPoint':
    return FixedPoint(-self.value, self.scale)

def __pos__(self) -> 'FixedPoint':
    return self
```

For the `__round__()`, `__trunc__()`, `__ceil__()`, and `__floor__()` operators, we've delegated the work to a Python library function. There are some potential optimizations, but we've taken the lazy route of creating a float approximation and used that to create the desired result. This suite of methods ensures that our `FixedPoint` objects will work with a number of arithmetic functions. Yes, there are a lot of operators in Python. This isn't the entire suite. We haven't provided implementations for the comparison or bit-kicking operators. The comparisons are generally similar to arithmetic operations, and are left as an exercise for the reader. The bit-wise operators (`&`, `|`, `^`, and `~`) don't have a clear meaning outside the domains, like values or sets, so we shouldn't implement them.

In the next section, we'll see how to implement `FixedPoint` reflected operators.

Implementing FixedPoint reflected operators

Reflected operators are used in the following two cases:

- The right-hand operand is a subclass of the left-hand operand. In this case, the reflected operator is tried first to ensure that the subclass overrides the parent class.
- The left-hand operand's class doesn't implement the special method required. In this case, the right-hand operand's reflected special method is used.

The following table shows the mapping between reflected special methods and operators:

Method	Operator
`object.__radd__(self, other)`	`+`
`object.__rsub__(self, other)`	`-`
`object.__rmul__(self, other)`	`*`
`object.__rtruediv__(self, other)`	`/`
`object.__rfloordiv__(self, other)`	`//`
`object.__rmod__(self, other)`	`%`
`object.__rdivmod__(self, other)`	`divmod()`

`object.__rpow__(self, other[, modulo])`	`pow()` as well as `**`

These reflected operation special methods are also built around a common boilerplate. Since these are reflected, the order of the operands in subtraction, division, modulus, and power is important. For commutative operations, such as addition and multiplication, the order doesn't matter. The following are the implementations for the reflected operators:

```
def __radd__(self, other: Union['FixedPoint', int]) -> 'FixedPoint':
    if not isinstance(other, FixedPoint):
        new_scale = self.scale
        new_value = other * self.scale + self.value
    else:
        new_scale = max(self.scale, other.scale)
        new_value = other.value * (new_scale // other.scale) + self.value * (
            new_scale // self.scale
        )
    return FixedPoint(int(new_value), scale=new_scale)

def __rsub__(self, other: Union['FixedPoint', int]) -> 'FixedPoint':
    if not isinstance(other, FixedPoint):
        new_scale = self.scale
        new_value = other * self.scale - self.value
    else:
        new_scale = max(self.scale, other.scale)
        new_value = other.value * (new_scale // other.scale) - self.value * (
            new_scale // self.scale
        )
    return FixedPoint(int(new_value), scale=new_scale)

def __rmul__(self, other: Union['FixedPoint', int]) -> 'FixedPoint':
    if not isinstance(other, FixedPoint):
        new_scale = self.scale
        new_value = other * self.value
    else:
        new_scale = self.scale * other.scale
        new_value = other.value * self.value
    return FixedPoint(int(new_value), scale=new_scale)

def __rtruediv__(self, other: Union['FixedPoint', int]) -> 'FixedPoint':
    if not isinstance(other, FixedPoint):
        new_value = self.scale * int(other / (self.value / self.scale))
    else:
        new_value = int((other.value / other.scale) / self.value)
    return FixedPoint(new_value, scale=self.scale)

def __rfloordiv__(self, other: Union['FixedPoint', int]) -> 'FixedPoint':
    if not isinstance(other, FixedPoint):
        new_value = self.scale * int(other // (self.value / self.scale))
    else:
```

```
            new_value = int((other.value / other.scale) // self.value)
        return FixedPoint(new_value, scale=self.scale)

    def __rmod__(self, other: Union['FixedPoint', int]) -> 'FixedPoint':
        if not isinstance(other, FixedPoint):
            new_value = other % (self.value / self.scale)
        else:
            new_value = (other.value / other.scale) % (self.value / self.scale)
        return FixedPoint(new_value, scale=self.scale)

    def __rpow__(self, other: Union['FixedPoint', int]) -> 'FixedPoint':
        if not isinstance(other, FixedPoint):
            new_value = other ** (self.value / self.scale)
        else:
            new_value = (other.value / other.scale) ** self.value / self.scale
        return FixedPoint(int(new_value) * self.scale, scale=self.scale)
```

We've tried to use math that is identical to the forward operators. The idea is to switch the operands in a simple way. This follows from the most common situation. Having the text of the forward and reverse methods match each other simplifies code inspections, and yes, there is some redundancy in the commutative operator implementations.

As with the forward operators, we've kept the division, modulus, and power operators simple to avoid optimizations. The versions shown here can introduce noise from the conversion to a floating-point approximation and back to a `FixedPoint` value. In the next section, we'll see how to implement `FixedPoint` comparison operators.

Implementing FixedPoint comparison operators

The following are the six comparison operators and the special methods that implement them:

Method	Operator
`object.__lt__(self, other)`	`<`
`object.__le__(self, other)`	`<=`
`object.__eq__(self, other)`	`==`
`object.__ne__(self, other)`	`!=`
`object.__gt__(self, other)`	`>`
`object.__ge__(self, other)`	`>=`

The `is` operator compares object IDs. We can't meaningfully override this since it's independent of any specific class. The `in` comparison operator is implemented by `object.__contains__(self, value)`. This isn't meaningful for numeric values.

Note that equality testing is a subtle business. Since floats are approximations, we have to be very careful to avoid direct equality testing with `float` values. We must compare to see whether the values are within a small range, that is, *epsilon*.

Equality tests should never be written as `a == b`. The general approach to compare floating-point approximations should be `abs(a-b) <= eps`, or, more generally, `abs(a-b)/a <= eps`.

In our `FixedPoint` class, the scale indicates how close two values need to be for a `float` value to be considered equal. For a scale of 100, the epsilon could be 0.01. We'll actually be more conservative than that and use 0.005 as the basis for comparison when the scale is 100.

Additionally, we have to decide whether `FixedPoint(123, 100)` should be equal to `FixedPoint(1230, 1000)`. While they're mathematically equal, one value is in cents and one is in mills.

This can be taken as a claim about the different accuracies of the two numbers; the presence of an additional significant digit may indicate that they're not supposed to simply appear equal. If we follow this approach, then we need to be sure that the hash values are different too.

For this example, we've decided that distinguishing among scale values is *not* appropriate. We want `FixedPoint(123, 100)` to be equal to `FixedPoint(1230, 1000)`. This is the assumption behind the recommended `__hash__()` implementation too. The following are the implementations for our `FixedPoint` class comparisons:

```
def __eq__(self, other: Any) -> bool:
    if isinstance(other, FixedPoint):
        if self.scale == other.scale:
            return self.value == other.value
        else:
            return self.value * other.scale // self.scale == other.value
    else:
        return abs(self.value / self.scale - float(other)) < .5 / self.scale

def __ne__(self, other: Any) -> bool:
    return not (self == other)

def __le__(self, other: 'FixedPoint') -> bool:
    return self.value / self.scale <= float(other)

def __lt__(self, other: 'FixedPoint') -> bool:
    return self.value / self.scale < float(other)

def __ge__(self, other: 'FixedPoint') -> bool:
    return self.value / self.scale >= float(other)
```

Each of the comparison functions tolerates a value that is not a `FixedPoint` value. This is a requirement imposed by the superclass: the `Any` type hint must be used

to be compatible with that class. The only requirement is that the other value must have a floating-point representation. We've defined a `__float__()` method for the `FixedPoint` objects, so the comparison operations will work perfectly well when comparing the two `FixedPoint` values.

We don't need to write all six comparisons. The `@functools.total_ordering` decorator can generate the missing methods from just two `FixedPoint` values. We'll look at this in `Chapter 9`, *Decorators and Mixins – Cross-Cutting Aspects*.

In the next section, we'll see how to compute a numeric hash.

Computing a numeric hash

We do need to define the `__hash__()` method properly. See section 4.4.4 of the *Python Standard Library* for information on computing hash values for numeric types. That section defines a `hash_fraction()` function, which is the recommended solution for what we're doing here. Our method looks like this:

```
def __hash__(self) -> int:
    P = sys.hash_info.modulus
    m, n = self.value, self.scale
    # Remove common factors of P.  (Unnecessary if m and n already coprime.)
    while m % P == n % P == 0:
        m, n = m // P, n // P

    if n % P == 0:
        hash_ = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: pow(n, P-1, P) is 1, so
        # pow(n, P-2, P) gives the inverse of n modulo P.
        hash_ = (abs(m) % P) * pow(n, P - 2, P) % P
    if m < 0:
        hash_ = -hash_
    if hash_ == -1:
        hash_ = -2
    return hash_
```

This reduces a two-part rational fraction value to a single, standardized hash. This code is copied with a few modifications from the reference manual. The core of the calculation, which is bolded in the preceding code, multiplies the numerator by the inverse of the denominator. In effect, it carries out the division of the numerator by the denominator, `mod P`. We can optimize this to make it more specific to our problem domain.

First, we could modify the `__new__()` method for this class to assure that the scale is non-zero, eliminating any need for `sys.hash_info.inf`. Second, we could explicitly limit the range of the scale factor to be less than `sys.hash_info.modulus` (generally $2^{61} - 1$ for 64-bit computers). We can eliminate the need to remove common factors of `P`. That would boil the hash down to `hash_ = (abs(m) % P) * pow(n, P - 2, P) % P`, sign handling, and the special case that -1 is mapped to -2.

Finally, we might want to cache the result of any hash calculation. This requires an additional slot that's only populated once, the first time a hash is requested. The `pow(n, P - 2, P)` expression is relatively expensive to evaluate and we don't

want to compute it more often than necessary.

In the next section, we'll show how to implement a simple rounding schema for these `FixedPoint` objects.

Designing more useful rounding

We truncated the presentation on rounding. We defined the required functions for `round()` and `trunc()` without further explanation. These definitions are the minimum requirements of the abstract superclass. However, these definitions are not quite enough for our purposes.

To process currency, we'll often have code that looks like this:

```
>>> price = FixedPoint(1299, 100)
>>> tax_rate = FixedPoint(725, 1000)
>>> price * tax_rate
FixedPoint(941775, scale=100000)
```

Then, we need to round this value to a scale of `100` to get a value of `942`. We need methods that will round (as well as truncate) a number to a new scale factor. The following is a method to round to a specific scale:

```
def round_to(self, new_scale: int) -> 'FixedPoint':
    f = new_scale / self.scale
    return FixedPoint(int(self.value * f + .5), scale=new_scale)
```

The following code allows us to properly rescale the value:

```
>>> price = FixedPoint(1299, 100)
>>> tax_rate = FixedPoint(725, 1000)
>>> tax = price * tax_rate
>>> tax.round_to(100)
FixedPoint(942, scale=100)
```

This shows that we have a minimal set of functions to calculate currency.

In the next section, we'll see how to implement other special methods.

Implementing other special methods

In addition to the core arithmetic and comparison operators, we have a group of additional operators that (generally) we only define for the `numbers.Integral` values. As we're not defining integral values, we can avoid these special methods:

Method	Operator
`object.__lshift__(self, other)`	`<<`
`object.__rshift__(self, other)`	`>>`
`object.__and__(self, other)`	`&`
`object.__xor__(self, other)`	`^`
`object.__or__(self, other)`	`|`

Also, there are reflected versions of these operators:

Method	Operator
`object.__rlshift__(self, other)`	`<<`
`object.__rrshift__(self, other)`	`>>`
`object.__rand__(self, other)`	`&`

object.__rxor__(self, other)	^
object.__ror__(self, other)	|

Additionally, there is a unary operator for a bit-wise inverse of the value:

Method	Operator
object.__invert__(self)	~

Interestingly, some of these operators are defined for the set collection, as well as integral numbers. They don't apply to our rational value. The principles to define these operators are the same as the other arithmetic operators.

Now, let's see how to optimize using the in-place operators.

Optimization with the in-place operators

Generally, numbers are immutable. However, the numeric operators are also used for mutable objects. Lists and sets, for example, respond to a few of the defined augmented assignment operators. As an optimization, a class can include an in-place version of a selected operator. The methods in the following table implement the augmented assignment statements for mutable objects. Note that these methods are expected to end with `return self` to be compatible with ordinary assignment:

Method	Operator
`object.__iadd__(self, other)`	`+=`
`object.__isub__(self, other)`	`-=`
`object.__imul__(self, other)`	`*=`
`object.__itruediv__(self, other)`	`/=`
`object.__ifloordiv__(self, other)`	`//=`
`object.__imod__(self, other)`	`%=`
`object.__ipow__(self, other[, modulo])`	`**=`
`object.__ilshift__(self, other)`	`<<=`

`object.__irshift__(self, other)`	`>>=`
`object.__iand__(self, other)`	`&=`
`object.__ixor__(self, other)`	`^=`
`object.__ior__(self, other)`	`|=`

As our `FixedPoint` objects are immutable, we should not define any of them.

Stepping outside this `FixedPoint` class example for a moment, we can see a more typical use for in-place operators. We could easily define some in-place operators for our Blackjack `Hand` objects. We might want to add this definition to `Hand` as follows:

```
def __iadd__(self, aCard):
    self._cards.append(aCard)
    return self
```

This allows us to deal into `hand` with the following code:

```
player_hand += deck.pop()
```

This seems to be an elegant way to indicate that `hand` is updated with another card.

Summary

We've looked at the built-in numeric types and the vast number of special methods required to invent a new numeric type. Specialized numeric types that integrate seamlessly with the rest of Python is one of the core strengths of the language. That doesn't make the job easy. It merely makes it elegant and useful when done properly.

When working with numbers, we have a multistep design strategy:

1. Consider the built-in versions of `complex`, `float`, and `int`.
2. Consider the library extensions, such as `decimal` and `fractions`. For financial calculations, `decimal` must be used; there is no alternative.
3. Consider extending one of the preceding classes with additional methods or attributes.
4. Finally, consider a novel number. This is particularly challenging since Python's variety of available numbers is already very rich.

Defining new numbers involves several considerations:

- **Completeness and consistency**: The new number must perform a complete set of operations and behave consistently in all kinds of expressions. This is really a question of properly implementing the formal mathematical definitions of this new kind of computable number.
- **Fit with the problem domain**: Is this number truly suitable? Does it help clarify the solution?
- **Performance**: As with other design questions, we must be sure that our implementation is efficient enough to warrant writing all that code. Our example in this chapter uses some inefficient floating-point operations that could be optimized by doing a little more math and a little less coding.

The next chapter is about using decorators and mixins to simplify and normalize class design. We can use decorators to define features that should be present in a number of classes, which are not in a simple inheritance hierarchy. Similarly, we can use mixin class definitions to create a complete application class from component class definitions. One of the decorators that is helpful for defining

comparison operators is the `@functools.total_ordering` decorator.

Decorators and Mixins - Cross-Cutting Aspects

A software design often has aspects that apply across several classes, functions, or methods. We might have a concern, such as logging, auditing, or security, that must be implemented consistently. One general method for reuse of functionality in object-oriented programming is inheritance through a class hierarchy. However, inheritance doesn't always work out well. For example, one aspect of a software design could be orthogonal to the class hierarchy. These are sometimes called **cross-cutting concerns**. They cut across the classes, making design more complex.

A decorator provides a way to define functionality that's not bound to the inheritance hierarchy. We can use decorators to design an aspect of our application and then apply the decorators across classes, methods, or functions.

Additionally, we can use multiple inheritances in a disciplined way to create cross-cutting aspects. We'll consider a base class plus mixin class definitions to introduce features. Often, we'll use the mixin classes to build cross-cutting aspects.

It's important to note that cross-cutting concerns are rarely specific to the application at hand. They're often generic considerations. The common examples of logging, auditing, and security could be considered as infrastructure separate from the application's details.

Python comes with many decorators, and we can expand this standard set of decorators. There are several different use cases. This chapter will start with a look at class definitions and the meaning of a class. With that context, we'll look at simple function decorators, function decorators with arguments, class decorators, and method decorators.

In this chapter, we will cover the following topics:

- Class and meaning

- Using built-in decorators
- Using standard library mixin classes
- Writing a simple function decorator
- Parameterizing a decorator
- Creating a method function decorator
- Creating a class decorator
- Adding methods to a class
- Using decorators for security

Technical requirements

The code files for this chapter are available at `https://git.io/fj2UV`.

Class and meaning

One essential feature of objects is that they can be classified: every object belongs to a class. This leads to a straightforward relationship between an object and class when using simple, single-inheritance design.

With multiple inheritance, the classification problem can become complex. When we look at real-world objects, such as coffee cups, we can classify them as containers without too much difficulty. That is, after all, their primary use case. The problem they solve is that of holding coffee. However, in another context, we may be interested in other use cases. Within a decorative collection of ceramic mugs, we might be more interested in size, shape, and glaze than we are in the coffee-carrying aspect of a cup.

Most objects have a straightforward *is-a* relationship with a class. In our coffee-holding problem domain, the mug sitting on the desk is in the class of coffee cups as well as the more general class of containers. Objects may also have several *acts-as* relationships with other classes. A mug acts as a piece of ceramic art with size, shape, and glaze properties. A mug acts as a paper weight with mass and friction properties.

Generally, these other features can be seen as mixin classes, and they define the additional interfaces or behaviors for an object. The mixin classes may have their own hierarchies; for example, ceramic art is a specialization of the more general sculpture and art classes.

When doing object-oriented design in Python, it's helpful to identify the *is-a* class and the essential aspects defined by that class. Other classes provide *acts-as* aspects, which mix in additional interfaces or behaviors for an object.

In the next section, we'll look at function definition and decoration because it's simpler than class construction. After looking at how function decoration works, we'll return to mixin classes and class decoration.

Type hints and attributes for decorators

We construct decorated functions in two stages. The first stage is the `def` statement with an original definition.

A `def` statement provides a name, parameters, defaults, a `docstring`, a code object, and a number of other details. A function is a collection of 11 attributes, defined in section 3.2 of the *Python Standard Library*, which is the standard type hierarchy.

The second stage involves the application of a decorator to the original definition. When we apply a decorator, (`@d`), to a function (`F`), the effect is as if we have created a new function, $F' = @d(F)$. The name, F, is the same, but the functionality can be different, depending on the kinds of features that have been added, removed, or modified. Generally, we can write the following code:

```
@decorate
def function():
    pass
```

The decorator is written immediately in front of the function definition. What happens to implement this can be seen by the following:

```
def function():
    pass
function = decorate(function)
```

The decorator modifies the function definition to create a new function. The essential technique here is that a decorator function accepts a function and returns a modified version of that function. Because of this, a decorator has a rather complex type hint.

This second style, `function=decorate(function)`, also works with functions created by assigning a lambda to a variable. It works with callable objects, also. The `@decorate` notation only works with the `def` statement.

When there are multiple decorators, they are applied as nested function calls.

Consider the following example:

```
@decorator1
@decorator2
def function(): ...
```

This is equivalent to `function=decorator1(decorator2(function))`. When decorators have side effects, the order of applying the decorations will matter. In the Flask framework, for example, the `@app.route` decoration should always be at the top of the stack of decorators so that it is applied last and includes the results of other decorators' behaviors.

The following is a typical set of type hints required to define a decorator:

```
from typing import Any, Callable, TypeVar, cast

FuncType = Callable[..., Any]
F = TypeVar('F', bound=FuncType)

def my_decorator(func: F) -> F:
    ...
```

We've defined a function type, `FuncType`, based on the `Callable` type hint. From this, the type variable, `F`, is derived as a generic description of anything that adheres to the `FuncType` protocol. This will include functions, lambdas, and callable objects. The decorator function, `my_decorator()`, accepts a parameter, `func`, with the type hint of `F`, and returns a function, using the type hint of `F`. What's essential is that any kind of object with the callable protocol can be described as having an upper boundary of the very generic `FuncType`. We've omitted the details of `my_decorator()` for now. This snippet is intended to show the general approach to type hints.

The decorator for a class is simpler, because the signature is `def class_decorator(class: Type) -> Type: ...`. There are a few ways to create classes, and the upper limit is already defined as the type hint `Type`.

Now, let's examine the different attributes of a function.

Attributes of a function

A decorator can change the attributes of a function. Here is the list of attributes of a function:

`__doc__`	The docstring, or none
`__name__`	The original name of the function
`__module__`	The name of the module the function was defined in, or none
`__qualname__`	The function's fully-qualified name, `__module__.__name__`
`__defaults__`	The default argument values, or none if there are no defaults
`__kwdefaults__`	The default values for keyword-only parameters
`__code__`	The code object representing the compiled function body
`__dict__`	A namespace for the function's attributes
`__annotations__`	The annotations of parameters, including `'return'` for the return annotation

`__globals__`	The global namespace of the module that the function was defined in; this is used to resolve global variables and is read-only
`__closure__`	Bindings for the function's free variables or none; it is read-only

Except for `__globals__` and `__closure__`, a decorator can change any of these attributes. As a practical matter, it's best to only copy the `__name__` and `__doc__` from the original function to the decorated function. Most of the other attributes, while changeable, are easier to manage with a simple technique of defining a new function inside the decorator and returning the new function. We'll look into this in the following several examples.

Now, let's see how to construct a decorated class.

Constructing a decorated class

A decorated class construction is a nested set of several two-stage processes. Making class construction more complex is the way references are made to class methods. The references involve a multistep lookup. An object's class will define a **Method Resolution Order** (**MRO**). This defines how base classes are searched to locate an attribute or method name. The MRO works its way up the inheritance hierarchy; this is how a subclass name can override a name in a superclass.

The outermost part of the nesting is processing the `class` statement as a whole. This has two stages: building the class, and applying the decorator functions. Within the `class` statement processing, the individual method definitions can also have decorations, and each of those is a two-stage process.

The first stage in class construction is the execution of the `class` statement. This stage involves the evaluation of the metaclass followed by the execution of the sequence of assignment and `def` statements within a `class`. Each `def` statement within the class expands to a nested two-stage function construction as described previously. Decorators can be applied to each method function as part of the process of building the class.

The second stage in class construction is to apply an overall class decorator to a class definition. Generally, this can add features. It's somewhat more common to add attributes rather than methods. While it is possible for decorators to add method functions, it can be hopelessly confusing for software maintainers to locate the source for a method injected by a decorator. These kinds of features need to be designed with considerable care.

The features inherited from the superclasses cannot be modified through decorators since they are resolved lazily by method resolution lookup. This leads to some important design considerations. We generally want to introduce methods and attributes through classes and mixin classes. We should limit ourselves to defining new attributes via decorators.

Here's a list of some of the attributes that are built for a class. A number of

additional attributes are part of the metaclass; they are described in the following table:

`__doc__`	The class's documentation string, or none if undefined
`__name__`	The class name
`__module__`	The module name that the class was defined in
`__dict__`	The dictionary containing the class's namespace
`__bases__`	A tuple (possibly empty or a singleton) containing the base classes, in the order of their occurrence in the base class list; it is used to work out the method resolution order
`__class__`	The superclass of this class, often type

Some additional method functions that are part of a class include `__subclasshook__`, `__reduce__`, and `__reduce_ex__`, which are part of the interface for `pickle`.

Some class design principles

When defining a class, we have the following sources of attributes and methods:

- Any decorators applied to the class definition. These are applied to the definition last.
- The body of the class statement.
- Any mixin classes. These definitions tend to override the base class definitions in the method resolution order algorithm.
- The base class. If unspecified, the base class is `object`, which provides a minimal set of definitions.

These are presented in order of their visibility. The final changes from a decorator overwrite everything below it, making these changes most visible. The body of the class statement overrides anything inherited from mixins or the base class. The base class is the last place used to resolve names.

We need to be cognizant about how easy it is for software maintainers to see each of these. The `class` statement is the most obvious place for someone to look for the definition of an attribute or methods. The mixins and the base class are somewhat less obvious than the class body. It's helpful to make sure that the base class name clarifies its role and uses terminology that is clearly essential. For example, it helps to name base classes after real-world objects.

The application of the decorator to the class can lead to obscure features. A strong focus on one or a few features helps to clarify what the decorator does. While some aspects of an application can be suitable for generic decorators, the lack of visibility can make them difficult to test, debug, and maintain.

The mixin classes will generally define additional interfaces or behaviors of a class. It's important to be clear about how the mixin classes are used to build the final class definitions. While a `docstring` class is an important part of this, the overall `docstring` module is also important to show how a proper class can be assembled from the various parts.

When writing the `class` statement, the mixins are listed first, and the essential

superclass is listed last. This is the search order for name resolution. The last listed class is the class that defines the essential *is-a* relationship. The last class on a list defines what a thing **IS**. The previous class names can define what a thing **DOES**. The mixins provide ways to override or extend this base behavior.

Aspect-oriented programming is discussed in the next section.

Aspect-oriented programming

Parts of **aspect-oriented programming** (**AOP**) are implemented by decorators in Python. Our purpose here is to leverage a few aspect-oriented concepts to help show the purpose of decorators and mixins in Python. The idea of a **cross-cutting concern** is central to AOP. While the Wikipedia page (`http://en.wikipedia.org/wiki/Cross-cutting_concern`) is generally kept up-to-date, older information is available here: `https://web.archive.org/web/20150919015041/http://www.aosd.net/wiki/index.php?title=Glossary`. The Spring framework provides some ideas; see also `https://docs.spring.io/spring-python/1.2.x/sphinx/html/aop.html`. There are several common examples of cross-cutting concerns, as follows:

- **Logging**: We often need to have logging features implemented consistently in many classes. We want to be sure that the loggers are named consistently and that logging events follow the class structure in a consistent manner.
- **Auditability**: A variation of the logging theme is to provide an audit trail that shows each transformation of a mutable object. In many commerce-oriented applications, the transactions are business records that represent bills or payments. Each step in the processing of a business record needs to be auditable to show that no errors have been introduced by processing.
- **Security**: Our applications will often have security aspects that pervade each HTTP request and each piece of content downloaded by the website. The idea is to confirm that each request involves an authenticated user who is authorized to make the request. Cookies, secure sockets, and other cryptographic techniques must be used consistently to assure that an entire web application is secured.

Some languages and tools have deep, formal support for AOP. Python borrows a few of the concepts. The Pythonic approach to AOP involves the following language features:

- **Decorators**: Using a decorator, we can establish a consistent aspect implementation at one of two simple join points in a function. We can perform the aspect's processing before or after the existing function. We can't easily locate join points inside the code of a function. It's easiest for decorators to transform a function or method by wrapping it with additional

functionality.

- **Mixins**: Using a mixin, we can define a class that exists as part of several class hierarchies. The mixin classes can be used with the base class to provide a consistent implementation of cross-cutting aspects. Generally, mixin classes are considered abstract, since they can't be meaningfully instantiated.

The next section shows how to use built-in decorators

Using built-in decorators

Python has several built-in decorators that are part of the language. The `@property`, `@classmethod`, and `@staticmethod` decorators are used to annotate methods of a class. The `@property` decorator transforms a method function into a descriptor. The `@property` decorator, when applied to a method function, changes the function into an attribute of the object. The property decorator, when applied to a method, also creates an additional pair of properties that can be used to create a `setter` and `deleter` property. We looked at this in `Chapter 4`, *Attribute Access, Properties, and Descriptors*.

The `@classmethod` and `@staticmethod` decorators transform a method function into a class-level function. The decorated method is now part of the class, not an object. In the case of a static method, there's no explicit reference to the class. With a class method, on the other hand, the class is the first argument of the method function. The following is an example of a class that includes `@staticmethod` and some `@property` definitions:

```
class Angle(float):
    __slots__ = ("_degrees",)

    @staticmethod
    def from_radians(value: float) -> 'Angle':
        return Angle(180 * value / math.pi)

    def __init__(self, degrees: float) -> None:
        self._degrees = degrees

    @property
    def radians(self) -> float:
        return math.pi * self._degrees / 180

    @property
    def degrees(self) -> float:
        return self._degrees
```

This class defines an `Angle` that can be represented in degrees or radians. The constructor expects degrees. However, we've also defined a `from_radians()` method function that emits an instance of the class. This function does not set values on an existing instance variable the way `__init__()` does; it creates a new instance of the class.

Additionally, we provide the `degrees()` and `radians()` method functions that have

been decorated so that they are properties. Under the hood, these decorators create a descriptor so that accessing the attribute name `degrees` or `radians` will invoke the named method function. We can use the `static` method to create an instance and then use a `property` method to access a method function as follows:

```
>>> b = Angle.from_radians(.227)
>>> round(b.degrees, 1)
13.0
```

The static method is similar to a function because it's not tied to the `self` instance variable. It has the advantage that it is syntactically bound to the class. Using `Angle.from_radians` can be more helpful than using a function named `angle_from_radians`. The use of these decorators ensures that implementation is handled correctly and consistently.

Now, let's see how to use standard library decorators.

Using standard library decorators

The standard library has a number of decorators. Modules such as `contextlib`, `functools`, `unittest`, `atexit`, `importlib`, and `reprlib` contain decorators that are excellent examples of cross-cutting aspects of a software design.

One particular example, the `functools` library, offers the `total_ordering` decorator that defines comparison operators. It leverages `__eq__()` and either `__lt__()`, `__le__()`, `__gt__()`, or `__ge__()` to create a complete suite of comparisons.

First, we'll need this class to fully define a playing card as follows:

```
from enum import Enum
class Suit(Enum):
    Clubs = "♣"
    Diamonds = "♦"
    Hearts = "♥"
    Spades = "♠"
```

This class provides the enumerated values for the suits of the playing cards.

The following is a variation on the `Card` class that defines just two comparisons:

```
import functools

@functools.total_ordering
class CardTO:
    __slots__ = ("rank", "suit")

    def __init__(self, rank: int, suit: Suit) -> None:
        self.rank = rank
        self.suit = suit

    def __eq__(self, other: Any) -> bool:
        return self.rank == cast(CardTO, other).rank

    def __lt__(self, other: Any) -> bool:
        return self.rank < cast(CardTO, other).rank

    def __str__(self) -> str:
        return f"{self.rank:d}{self.suit:s}"
```

Our class, `CardTO`, is wrapped by a class-level decorator, `@functools.total_ordering`. This decorator creates the missing method functions to be sure all comparisons work. From some combinations of operators, the remainder can be derived. The general idea is to provide some form of equality (or inequality) test, and an

ordering test, and the remainder of the operations can be derived logically from those two.

In the example, we provided < and =. Here's how the other comparisons can be derived:

$$a < b : \textbf{given}$$
$$a = b : \textbf{given}$$
$$a \le b \equiv (a < b) \lor (a = b)$$
$$a > b \equiv \neg(a < b) \land \neg(a = b)$$
$$a \ge b \equiv \neg(a < b)$$
$$a \ne b \equiv \neg(a = b)$$

We can use this class to create objects that can be compared using all of the comparison operators, even though only two were defined as follows:

```
>>> c1 = Card( 3, '♠' )
>>> c2 = Card( 3, '♥' )
>>> c1 == c2
True
>>> c1 < c2
False
>>> c1 <= c2
True
>>> c1 >= c2
True
```

This interaction shows that we are able to make comparisons that are not defined in the original class. The decorator added the required method functions to the original class definition.

Let's see how to use standard library mixin classes in the next section.

Using standard library mixin classes

The standard library makes use of mixin class definitions. There are several modules that contain examples, including `io`, `socketserver`, `urllib.request`, `contextlib`, and `collections.abc`. Next, we'll look at an example using mixing features of the `Enum` class in the `enum` module.

When we define our own collection based on the `collections.abc` abstract base classes, we're making use of mixins to ensure that cross-cutting aspects of the containers are defined consistently. The top-level collections (`Set`, `Sequence`, and `Mapping`) are all built from multiple mixins. It's very important to look at section 8.4 of the *Python Standard Library* to see how the mixins contribute features as the overall structure is built up from pieces.

Looking at just one line, the summary of `Sequence`, we see that it inherits from `Sized`, `Iterable`, and `Container`. These mixin classes lead to methods of `__contains__()`, `__iter__()`, `__reversed__()`, `index()`, and `count()`.

The final behavior of the `list` class is a composition of aspects from each of the mixins present in its definition. Fundamentally, it's a `Container` with numerous protocols added to it.

Let's look at how to use the enum with mixin classes in the next section.

Using the enum with mixin classes

The `enum` module provides the `Enum` class. One common use case for this class is to define an enumerated domain of values; for example, we might use this to enumerate the four suits for playing cards.

An enumerated type has the following two features:

- **Member names:** The member names are proper Python identifiers for the enumerated values.
- **Member values:** The member values can be any Python object.

In several previous examples, we've used a simplistic definition for the enumerated members. Here's a typical class definition:

```
from enum import Enum
class Suit(Enum):
    Clubs = "♣"
    Diamonds = "♦"
    Hearts = "♥"
    Spades = "♠"
```

This provides four members. We can use `Suit.Clubs` to reference a specific string. We can also use `list(Suit)` to create a list of the enumerated members.

The base `Enum` class imposes constraints on the member names or values that will be part of the class. We can narrow the definition using mixin class definitions. Specifically, the Enum class can work with a data type as well as additional feature definitions.

We often want a richer definition of the underlying values for the members of the enumeration. This example shows the mixing of `str` into `Enum`:

```
class SuitS(str, Enum):
    Clubs = "♣"
    Diamonds = "♦"
    Hearts = "♥"
    Spades = "♠"
```

The base class is `Enum`. Features of the `str` class will be available to each member. The order of the definitions is important: the mixins are listed first; the base class

is listed last.

When `str` is mixed in, it provides all of the string methods to the member itself without having to make an explicit reference to the internal `value` of each member. For example, `SuitS.Clubs.center(5)` will emit the string value centered in a string of length five.

We can also incorporate additional features in an `Enum`. In this example, we'll add a class-level feature to enumerate the values:

```
class EnumDomain:
    @classmethod
    def domain(cls: Type) -> List[str]:
        return [m.value for m in cls]

class SuitD(str, EnumDomain, Enum):
    Clubs = "♣"
    Diamonds = "♦"
    Hearts = "♥"
    Spades = "♠"
```

The following two mixin protocols are added to this class:

- The str methods will apply to each member directly.
- The class also has a `domain()` method, which will emit only the values. We can use `SuitD.domain()` to get the list of string values associated with the members.

This mixin technique allows us to bundle features together to create complex class definitions from separate aspects.

TIP *A mixin design is better than copy and paste among several related classes.*

It can be difficult to create classes that are generic enough to be used as mixins. One approach is to look for duplicated copypasta code across multiple classes. The presence of duplicated code is an indication of a possible mixin to refactor and eliminate the duplication.

Let's see how to write a simple function decorator in the next section.

Writing a simple function decorator

A `decorator` is a function (or a callable object) that accepts a function as an argument and returns a new function. The result of decoration is a function that has been wrapped. Generally, the additional features of the wrapping surround the original functionality, either by transforming actual argument values or by transforming the result value. These are the two readily available join points in a function.

When we use a decorator, we want to be sure that the resulting decorated function has the original function's name and `docstring`. These details can be handled for us by a decorator to build our decorators. Using `functools.wraps` to write new decorators simplifies the work we need to do because the bookkeeping is handled for us.

Additionally, the type hints for decorators can be confusing because the parameter and return are both essentially of the `Callable` type. To be properly generic, we'll use an upper-bound type definition to define a type, `F`, which embraces any variation on callable objects or functions.

To illustrate the two places where functionality can be inserted, we can create a debug trace decorator that will log parameters and return values from a function. This puts functionality both before and after the called function. The following is a defined function, `some_function`, that we want to wrap. In effect, we want code that behaves like the following:

```
logging.debug("function(%r, %r)", args, kw)
result = some_function(*args, **kw)
logging.debug("result = %r", result)
```

This snippet shows how we'll have new log-writing to wrap the original, `some_function()`, function.

The following is a debug decorator that inserts logging before and after function evaluation:

```
import logging, sys
import functools
```

```
from typing import Callable, TypeVar

FuncType = Callable[..., Any]
F = TypeVar('F', bound=FuncType)

def debug(function: F) -> F:

    @functools.wraps(function)
    def logged_function(*args, **kw):
        logging.debug("%s(%r, %r)", function.__name__, args, kw)
        result = function(*args, **kw)
        logging.debug("%s = %r", function.__name__, result)
        return result

    return cast(F, logged_function)
```

We've used the `@functools.wraps` decorator to ensure that the original function name and docstring are preserved as attributes of the result function. The `logged_function()` definition is the resulting function returned by the `debug()` decorator. The internal, `logged_function()` does some logging, then invokes the decorated function, `function`, and does some more logging before returning the result of the decorated function. In this example, no transformation of argument values or results was performed.

When working with the logger, f-strings are not the best idea. It can help to provide individual values so the logging filters can be used to redact or exclude entries from a sensitive log.

Given this `@debug` decorator, we can use it to produce noisy, detailed debugging. For example, we can do this to apply the decorator to a function, `ackermann()`, as follows:

```
@debug
def ackermann(m: int, n: int) -> int:
    if m == 0:
        return n + 1
    elif m > 0 and n == 0:
        return ackermann(m - 1, 1)
    elif m > 0 and n > 0:
        return ackermann(m - 1, ackermann(m, n - 1))
    else:
        raise Exception(f"Design Error: {vars()}")
```

This definition wraps the `ackermann()` function with debugging information written via the logging module to the `root` logger. We've made no material changes to the function definition. The `@debug` decorator injects the logging details as a separate aspect.

We configure the logger as follows:

```
logging.basicConfig(stream=sys.stderr, level=logging.DEBUG)
```

We'll revisit logging in detail in `Chapter 16`, *The Logging and Warning Modules*. We'll see this kind of result when we evaluate `ackermann(2,4)` as follows:

```
DEBUG:root:ackermann((2, 4), {})
DEBUG:root:ackermann((2, 3), {})
DEBUG:root:ackermann((2, 2), {})
.
.
.
DEBUG:root:ackermann((0, 10), {})
DEBUG:root:ackermann = 11
DEBUG:root:ackermann = 11
DEBUG:root:ackermann = 11
```

In the next section, we will see how to create separate loggers.

Creating separate loggers

As a logging optimization, we might want to use a specific logger for each wrapped function and not overuse the root logger for this kind of debugging output. We'll return to the logger in `Chapter 16`, *The Logging and Warning Modules*.

The following is a version of our decorator that creates a separate logger for each individual function:

```
def debug2(function: F) -> F:
    log = logging.getLogger(function.__name__)

    @functools.wraps(function)
    def logged_function(*args, **kw):
        log.debug("call(%r, %r)", args, kw)
        result = function(*args, **kw)
        log.debug("result = %r", result)
        return result

    return cast(F, logged_function)
```

This version modifies the output to look like the following:

```
DEBUG:ackermann:call((2, 4), {})
DEBUG:ackermann:call((2, 3), {})
DEBUG:ackermann:call((2, 2), {})
.
.
.
DEBUG:ackermann:call( (0, 10), {} )
DEBUG:ackermann:result = 11
DEBUG:ackermann:result = 11
DEBUG:ackermann:result = 11
```

The function name is now the logger name. This can be used to fine-tune the debugging output. We can now enable logging for individual functions rather than enabling debugging for all functions.

Note that we can't trivially change the decorator and expect the decorated function to also change. After making a change to a decorator, we need to apply the revised decorator to a function. This means that debugging and experimenting with decorators can't be done *trivially* from the `>>>` interactive prompt.

Decorator development often involves creating and rerunning a script to define the decorator and apply it to example functions. In some cases, this script will also include tests or a demonstration to show that everything works as expected.

Now, let's see how to parameterize a decorator.

Parameterizing a decorator

Sometimes, we need to provide parameters to a decorator. The idea is that we are going to customize the wrapping function. When we do this, decoration becomes a two-step process.

Here's a snippet showing how we provide a parameterized decorator to a function definition:

```
@decorator(arg)
def func( ):
    pass
```

The implementation is as follows:

```
def func( ):
    pass
func = decorator(arg)(func)
```

We've done the following three things:

- Defined a function, `func`
- Applied the abstract decorator to its arguments to create a concrete decorator, `decorator(arg)`
- Applied the concrete decorator to the defined function to create the decorated version of the function, `decorator(arg)(func)`

It can help to think of `func = decorate(arg)(func)` as having the following implementation:

```
concrete = decorate(arg)
func = concrete(func)
```

This means that a decorator with arguments is implemented as indirect construction of the final function. Now, let's tweak our debugging decorator yet again. We'd like to do the following:

```
@debug("log_name")
def some_function( args ):
    pass
```

This kind of code allows us to specify the name of the log that the debugging output will go to. This means we won't use the root logger or create a distinct logger for each function.

The outline of a parameterized decorator will be the following:

```
def decorator(config) -> Callable[[F], F]:
    def concrete_decorator(function: F) -> F:
        def wrapped(*args, **kw):
            return function(*args, **kw)
        return cast(F, wrapped)
    return concrete_decorator
```

Let's peel back the layers of this onion before looking at the example. The decorator definition (`def decorator(config)`) shows the parameters we will provide to the decorator when we use it. The body of this is the concrete decorator, which is returned after the parameters are bound to it. The concrete decorator (`def concrete_decorator(function):`) will then be applied to the target function. The concrete decorator is like the simple function decorator shown in the previous section. It builds the wrapped function (`def wrapped(*args, **kw):`), which it returns.

The following is our named logger version of debug:

```
def debug_named(log_name: str) -> Callable[[F], F]:
    log = logging.getLogger(log_name)

    def concrete_decorator(function: F) -> F:

        @functools.wraps(function)
        def wrapped(*args, **kw):
            log.debug("%s(%r, %r)", function.__name__, args, kw)
            result = function(*args, **kw)
            log.debug("%s = %r", function.__name__, result)
            return result

        return cast(F, wrapped)

    return concrete_decorator
```

This `@debug_named` decorator accepts an argument that is the name of the log to use. It creates and returns a concrete decorator function with a logger of the given name bound into it. When this concrete decorator is applied to a function, the concrete decorator returns the wrapped version of the given function. When the function is used in the following manner, the decorator adds noisy debug lines.

Here's an example of creating a logged named recursion with output from a given function:

```python
@debug_named("recursion")
def ackermann3(m: int, n: int) -> int:
    if m == 0:
        return n + 1
    elif m > 0 and n == 0:
        return ackermann3(m - 1, 1)
    elif m > 0 and n > 0:
        return ackermann3(m - 1, ackermann3(m, n - 1))
    else:
        raise Exception(f"Design Error: {vars()}")
```

The decorator wraps the given `ackermann3()` function with logging output. Since the decorator accepts a parameter, we can provide a logger name. We can reuse the decorator to put any number of individual functions into a single logger, providing more control over the debug output from an application.

Now, let's see how to create a method function decorator.

Creating a method function decorator

A decorator for a method of a class definition is identical to a decorator for a standalone function. While it's used in a different context, it will be defined like any other decorator. One small consequence of the different context is that we often, must explicitly name the `self` variable in decorators intended for methods.

One application for method decoration is to produce an audit trail for object state changes. Business applications often create stateful records; commonly, these are represented as rows in a relational database. We'll look at object representation in `Chapter 10`, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML*, `Chapter 11`, *Storing and Retrieving Objects via Shelve*, and `Chapter 12`, *Storing and Retrieving Objects via SQLite*.

When we have stateful records, the state changes often need to be auditable. An audit can confirm that appropriate changes have been made to the records. In order to do the audit, the before and after version of each record must be available somewhere. Stateful database records are a long-standing tradition but are not in any way required. Immutable database records are a viable design alternative.

When we design a stateful class, any setter method will cause a state change. If we do this, we can fold in an `@audit` decorator that can track changes to the object so that we have a proper trail of changes. We'll create an audit log via the `logging` module. We'll use the `__repr__()` method function to produce a complete text representation that can be used to examine changes.

The following is an audit decorator:

```
def audit(method: F) -> F:

    @functools.wraps(method)
    def wrapper(self, *args, **kw):
        template = "%s\n     before %s\n     after %s"
        audit_log = logging.getLogger("audit")
        before = repr(self)  # preserve state as text
        try:
            result = method(self, *args, **kw)
        except Exception as e:
            after = repr(self)
            audit_log.exception(template, method.__qualname__, before, after)
            raise
        after = repr(self)
        audit_log.info(template, method.__qualname__, before, after)
        return result
```

```
    return cast(F, wrapper)
```

This audit trail works by creating text mementos of the before setting and after setting state of the object. After capturing the `before` state, the original method function is applied. If there was an exception, an audit log entry includes the exception details. Otherwise, an `INFO` entry is written with the qualified name of the method name, the `before` memento, and the `after` memento of the object.

The following is a modification of the `Hand` class that shows how we'd use this decorator:

```
class Hand:

    def __init__(self, *cards: CardDC) -> None:
        self._cards = list(cards)

    @audit
    def __iadd__(self, card: CardDC) -> "Hand":
        self._cards.append(card)
        self._cards.sort(key=lambda c: c.rank)
        return self

    def __repr__(self) -> str:
        cards = ", ".join(map(str, self._cards))
        return f"{self.__class__.__name__}({cards})"
```

This definition modifies the `__iadd__()` method function so that adding a card becomes an auditable event. This decorator will perform the audit operation, saving text mementos of `Hand` before and after the operation.

This use of a method decorator makes a visible declaration that a particular method will make a significant state change. We can easily use code reviews to be sure that all of the appropriate method functions are marked for audit like this.

In the event that we want to audit object creation as well as state change, we can't use this `audit` decorator on the `__init__()` method function. That's because there's no before image prior to the execution of `__init__()`. There are two things we can do as a remedy to this, as follows:

- We can add a `__new__()` method that ensures that an empty `_cards` attribute is seeded into the class as an empty collection.
- We can tweak the `audit()` decorator to tolerate `AttributeError` that will arise when `__init__()` is being processed.

The second option is considerably more flexible. We can do the following:

```
try:
    before = repr(self)
except AttributeError as e:
    before = repr(e)
```

This would record a message such as `AttributeError: 'Hand' object has no attribute '_cards'` for the `before` status during initialization.

In the next section, we'll see how to create a class decorator.

Creating a class decorator

Analogous to decorating a function, we can write a class decorator to add features to a class definition. The essential rules are the same. The decorator is a function (or callable object); it receives a class object as an argument and returns a class object as a result.

We have a limited number of join points inside a class definition as a whole. For the most part, a class decorator can fold additional attributes into a class definition. While it's technically possible to create a new class that wraps an original class definition, this doesn't seem to be very useful as a design pattern. It's also possible to create a new class that is a subclass of the original decorated class definition. This may be baffling to users of the decorator. It's also possible to delete features from a class definition, but this seems perfectly awful.

One sophisticated class decorator was shown previously. The `functools.Total_Ordering` decorator injects a number of new method functions into the class definition. The technique used in this implementation is to create lambda objects and assign them to attributes of the class.

In general, adding attributes often leads to problems with **mypy** type hint checking. When we add attributes to a class in a decorator, they're essentially invisible to **mypy**.

As an example, consider the need to debug object creation. Often, we'd like to have a unique logger for each class.

We're often forced to do something like the following:

```
class UglyClass1:

    def __init__(self) -> None:
        self.logger = logging.getLogger(self.__class__.__qualname__)
        self.logger.info("New thing")

    def method(self, *args: Any) -> int:
        self.logger.info("method %r", args)
        return 42
```

This class has the disadvantage that it creates a `logger` instance variable that's

really not part of the class's operation, but is a separate aspect of the class. We'd like to avoid polluting the class with this additional aspect. Even though `logging.getLogger()` is very efficient, the cost's non-zero. We'd like to avoid this additional overhead every time we create an instance of `UglyClass1`.

Here's a slightly better version. The logger is promoted to be a class-level instance variable and is separate from each individual object of the class:

```
class UglyClass2:
    logger = logging.getLogger("UglyClass2")

    def __init__(self) -> None:
        self.logger.info("New thing")

    def method(self, *args: Any) -> int:
        self.logger.info("method %r", args)
        return 42
```

This has the advantage that it implements `logging.getLogger()` just once. However, it suffers from a profound **Don't Repeat Yourself** (**DRY**) problem. We can't automatically set the class name within the class definition. The class hasn't been created yet, so we're forced to repeat the name.

The DRY problem can be partially solved by a small decorator as follows:

```
def logged(class_: Type) -> Type:
    class_.logger = logging.getLogger(class_.__qualname__)
    return class_
```

This decorator tweaks the class definition to add the `logger` reference as a class-level attribute. Now, each method can use `self.logger` to produce audit or debug information. When we want to use this feature, we can use the `@logged` decorator on the class as a whole.

This presents a profound problem for **mypy**, more easily solved with a mixin than a decorator.

Continuing to use the class decorator, the following is an example of a logged class, `SomeClass`:

```
@logged
class SomeClass:

    def __init__(self) -> None:
        self.logger.info("New thing")  # mypy error
```

```
    def method(self, *args: Any) -> int:
        self.logger.info("method %r", args)  # mypy error
        return 42
```

The decorator guarantees that the class has a `logger` attribute that can be used by any method. The `logger` attribute is not a feature of each individual instance, but a feature of the class as a whole. This attribute has the added benefit that it creates the logger instances during module import, reducing the overhead of logging slightly. Let's compare this with `UglyClass1`, where `logging.getLogger()` was evaluated for each instance creation.

We've annotated two lines that will report **mypy** errors. The type hint checks whether attributes injected by decorators are not robust enough to detect the additional attribute. The decorator can't easily create attributes visible to **mypy**. It's better to use the following kind of mixin:

```
class LoggedWithHook:
    def __init_subclass__(cls, name=None):
        cls.logger = logging.getLogger(name or cls.__qualname__)
```

This mixin class defines the `__init_subclass__()` method to inject an additional attribute into the class definition. This is recognized by **mypy**, making the `logger` attribute visible and useful. If the name of the parameter is provided, it becomes the name of the logger, otherwise the name of the subclass will be used. Here's an example class making use of this mixin:

```
class SomeClass4(LoggedWithHook):

    def __init__(self) -> None:
        self.logger.info("New thing")

    def method(self, *args: Any) -> int:
        self.logger.info("method %r", args)
        return 42
```

This class will have a logger built when the class is created. It will be shared by all instances of the class. And the additional attribute will be visible to **mypy**. In most ordinary application programming, class-level decorators are a rarity. Almost anything needed can be done using the `__init_subclass__()` method.

Some complex frameworks, such as the `@dataclasses.dataclass` decorator, involve extending the class from the available scaffolding. The code required to introduce names into the attributes used by **mypy** is unusual.

Let's see how to add methods to a class in the next section.

Adding methods to a class

A class decorator can create new methods using a two-step process. First, it must create a method function and then insert it into the class definition. This is often better done via a mixin class than a decorator. The obvious and expected use of a mixin is to insert methods. Inserting methods with a decorator is less obvious and can be astonishing to people reading the code and trying to find where the methods of a class are defined.

In the example of the `Total_Ordering` decorator, the exact method functions inserted were flexible and depended on what was already provided. This was a kind of special case that doesn't tend to astonish people reading the code.

We'll look at a technique to create a snapshot of an object's state by creating a text *memento* of the object. This can be implemented via a standardized `memento()` method. We'd like to include this standard method function in a variety of classes. First, we'll look at a decorator implementation. After that, we'll look at a mixin version of this design.

The following is the decorator version of adding this standardized `memento()` method:

```
def memento(class_: Type) -> Type:

    def memento_method(self):
        return f"{self.__class__.__qualname__}(**{vars(self)!r})"

    class_.memento = memento_method
    return class_
```

This decorator includes a method function definition that is inserted into the class. The `vars(self)` expression exposes the instance variables usually kept the internal `__dict__` attribute of an instance. This produces a dictionary that can be included in the output string value.

The following is how we use this `@memento` decorator to add the `memento()` method to a class:

```
@memento
class StatefulClass:
```

```
    def __init__(self, value: Any) -> None:
        self.value = value

    def __repr__(self) -> str:
        return f"{self.value}"
```

The decorator incorporates a new method, `memento()`, into the decorated class. Here's an example of using this class and extracting a memento that summarizes the state of the object:

```
>>> st = StatefulClass(2.7)
>>> print(st.memento())
StatefulClass(**{'value': 2.7})
```

This implementation has the following disadvantages:

- We can't override the implementation of the `memento()` method function to handle special cases. It's built into the class *after* the definition.
- We can't extend the decorator function easily. Doing this would involve creating either a very complex `memento()` method, or perhaps some other unwieldy design to incorporate some kind of plug-in feature.

An alternative is to use a mixin class. Variations on this class allow customization. The following is the mixin class that adds a standard method:

```
class Memento:

    def memento(self) -> str:
        return (
            f"{self.__class__.__qualname__}"
            f"(**{vars(self)!r})"
        )
```

The following is how we use this `Memento` mixin class to define an application class:

```
class StatefulClass2(Memento):

    def __init__(self, value):
        self.value = value

    def __repr__(self):
        return f"{self.value}"
```

The mixin provides a new method, `memento()`; this is the expected, typical purpose of a mixin. We can more easily extend the `Memento` mixin class to add features. In addition, we can override the `memento()` method function to handle special cases.

Now, let's see how to use decorators for security.

Using decorators for security

Software is filled with cross-cutting concerns, aspects that need to be implemented consistently, even if they're in separate class hierarchies. It's often a mistake to try and impose a class hierarchy around a cross-cutting concern. We've looked at a few examples, such as logging and auditing.

We can't reasonably demand that every class that might need to write to the log also be a subclass of some single `Loggable` superclass. It's much easier to design a `Loggable` mixin or a `@loggable` decorator. These don't interfere with the proper inheritance hierarchy that we need to design to make polymorphism work correctly.

Some important cross-cutting concerns revolve around security. Within a web application, there are two aspects to the security question as follows:

- **Authentication**: Do we know who's making the request?
- **Authorization**: Is the authenticated user permitted to make the request?

Some web frameworks allow us to decorate our request handlers with security requirements. The Django framework, for example, has a number of decorators that allow us to specify security requirements for a view function or a view class.

Some of these decorators are as follows:

- `user_passes_test`: This is a low-level decorator that's very generalized and is used to build the other two decorators. It requires a test function; the logged-in `User` object associated with the request must pass the given function. If the `User` instance is not able to pass the given test, they're redirected to a login page so that the person can provide the credentials required to make the request.
- `login_required`: This decorator is based on `user_passes_test`. It confirms that the logged-in user is authenticated. This kind of decorator is used on web requests that apply to all people accessing the site. Requests, such as changing a password or logging out, shouldn't require any more specific permissions.

- `permission_required`: This decorator works with Django's internally defined database permission scheme. It confirms that the logged-in user (or the user's group) is associated with the given permission. This kind of decorator is used on web requests where specific administrative permissions are required to make the request.

Other packages and frameworks also have ways to express this cross-cutting aspect of web applications. In many cases, a web application may have even more stringent security considerations. We might have a web application where user features are selectively unlocked based on contract terms and conditions. Perhaps, additional fees will unlock a feature. We might have to design a test like the following:

```
def user_has_feature(feature_name):
    def has_feature(user):
        return feature_name in (f.name for f in user.feature_set())
    return user_passes_test(has_feature)
```

This decorator customizes a version of the Django `user_passes_test()` decorator by binding in a specific feature test. The `has_feature()` function checks a `feature_set()` value of each `User` object. This is not built-in in Django. The `feature_set()` method would be an extension, added onto the Django `User` class definition. The idea is for an application to extend the Django definitions to define additional features.

The `has_feature()` function checks to see whether the named feature is associated with the `feature_set()` results for the current `User` instance. We've used our `has_feature()` function with Django's `user_passes_test` decorator to create a new decorator that can be applied to the relevant `view` functions.

We can then create a `view` function as follows:

```
@user_has_feature('special_bonus')
def bonus_view(request):
    pass
```

This ensures that the security concerns will be applied consistently across a number of `view` functions.

Summary

We've looked at using decorators to modify function and class definitions. We've also looked at mixins that allow us to decompose a larger class into components that are knitted together.

The idea of both of these techniques is to separate application-specific features from generic features, such as security, audit, or logging. We're going to distinguish between the inherent features of a class and aspects that aren't inherent but are additional concerns. The inherent features are part of the explicit design. They're part of the inheritance hierarchy; they define what an object is. The other aspects can be mixins or decorations; they define how an object might also act.

In most cases, this division between *is-a* and *acts-as* is quite clear. Inherent features are a part of the overall problem domain. When talking about simulating Blackjack play, things such as cards, hands, betting, hitting, and standing are clearly part of the problem domain. Similarly, the data collection and statistical analysis of outcomes is part of the solution. Other things, such as logging, debugging, security checks, and auditing are not part of the problem domain; these other aspects are associated with the solution technology. In some cases, they are part of regulatory compliance or another background context in which the software is used.

While most cases are quite clear, the dividing line between inherent and decorative aspects can be fine. In some cases, it may devolve to an aesthetic judgment. Generally, the decision becomes difficult when writing framework and infrastructure classes because they aren't focused on a specific problem. A general strategy for creating good designs is as follows:

- Aspects that are central to the problem will contribute directly to class definitions. Many classes are based on nouns and verbs present in the problem domain. These classes form simple hierarchies; polymorphism among data objects works as expected when compared with real-world objects.
- Some aspects are peripheral to the problem and will lead to mixin class

definitions. These are things related to operational aspects of using the software more than solving the essential problem.

A class that involves mixins can be said to be *multidimensional*. It has more than one independent axis; aspects belong to orthogonal design considerations. When we define separate mixins, we can have separate inheritance hierarchies for the mixins. For our casino game simulations, there are two aspects: the rules of the game and a betting strategy. These are orthogonal considerations. The final player simulation classes must have mixin elements from both class hierarchies.

The type hints for decorators can become complex. In the most generic case, a decorator can be summarized as a function with an argument that's a `Callable` and a result that's a `Callable`. If we want to be specific about the arguments and results of the callable, there will be complex-looking type hints, often involving type variables to show how the `Callable` argument and the `Callable` result align. This can become very complex if the decorator changes the signature of the decorated function by modifying parameters or results.

As noted previously, object-oriented programming lets us follow a variety of design strategies, as follows:

- Composition: We introduce functionality through *wrapping* one class with another class. This may involve the composition of various aspects under a façade. It may involve using mixins classes to add features, or decorators to add features.
- Extension: This is the ordinary case of inheritance. This is appropriate where there is a clear *is-a* relationship among the class definitions. It works out best when the superclass is a unsurprising generalization of the subclass details. In this case, ordinary inheritance techniques work out well.

The forthcoming chapters will change direction. We've seen almost all of Python's special method names. The next five chapters are going to focus on object persistence and serialization. We'll start out with serializing and saving objects in various external notations, including JSON, YAML, Pickle, CSV, and XML.

Serialization and persistence introduce yet more object-oriented design considerations for our classes. We'll also have a look at object relationships and

how they're represented. We'll also have a look at the cost complexity of serializing and deserializing objects, and at the security issues related to the deserialization of objects from untrustworthy sources.

Section 2: Object Serialization and Persistence

A persistent object has been serialized to a storage medium. Perhaps it has been transformed to JSON and written to the filesystem. Perhaps an **object-relational management** (**ORM**) layer can store the object in a database.

The following chapters will be covered in this section:

Serializing and Saving - JSON, YAML, Pickle, CSV, and XML

To make a Python object persistent, we must convert it into bytes and write the bytes to a file. We'll call this transformation **serialization**; it is also called marshaling, deflating, or encoding. We'll look at several ways to serialize a Python object to a stream of bytes. It's important to note that we're focused on representing the state of an object, separate from the full definition of the class and its methods and superclasses.

A serialization scheme includes a **physical data format**. Each format offers some advantages and disadvantages. There's no *best* format to represent the state of objects. helps to distinguish the format from the **logical data layout**, which may be a simple reordering or change in the use of whitespace; the layout changes don't change the value of the object but change the sequence of bytes in an irrelevant way. For example, the CSV physical format can have a variety of logical layouts and still represent the same essential data. If we provide unique column titles, the order of the columns doesn't matter.

Some serialized representations are biased toward representing a single Python object, while others can save collections of individual objects. Even when the single object is a `list` of items, it's still a single Python object. In order to update or replace one of the items within the list, the entire list must be de-serialized and re-serialized. When it becomes necessary to work with multiple objects flexibly, there are better approaches described in `Chapters 11`, *Storing and Retrieving Objects via Shelve*, `Chapter 12`, *Storing and Retrieving Objects via SQLite*, and `Chapter 13`, *Transmitting and Sharing Objects.*

For the most part, we're limited to objects that fit in working memory. We'll look at the following serialization representations:

- **JavaScript Object Notation (JSON)**: This is a widely used representation. For more information, visit `http://www.json.org`. The `json` module provides the classes and functions necessary to load and dump data in this format. In the Python Standard Library, look at section 19, *Internet Data Handling*, not

section 12, *Persistence*. The `json` module is focused narrowly on JSON serialization. The more general problem of serializing arbitrary Python objects isn't handled well.

- **YAML Ain't Markup Language (YAML)**: This is an extension to JSON and can lead to some simplification of the serialized output. For more information, check out `http://yaml.org`. This is not a standard part of the Python library; we must add a module to handle this. The `PyYaml` package, specifically, has numerous Python persistence features.
- **pickle**: The `pickle` module has its own unique representation for data. As this is a first-class part of the Python library, we'll look closely at how to serialize an object in this way. This has the disadvantage of being a poor format for the interchange of data with non-Python programs. It's the basis for the `shelve` module in `Chapter 11`, *Storing and Retrieving Objects via Shelve*, as well as message queues in `Chapter 13`, *Transmitting and Sharing Objects*.
- **Comma Separated Values (CSV)**: This can be inconvenient for representing complex Python objects. As it's so widely used, we'll need to work out ways to serialize Python objects in the CSV notation. For references, look at section 14, *File Formats*, of the *Python Standard Library*, not section 12, *Persistence*, because it's simply a file format and little more. CSV allows us to perform an incremental representation of the Python object collections that cannot fit into memory.
- **XML**: In spite of some disadvantages, this is very widely used, so it's important to be able to convert objects into an XML notation and recover objects from an XML document. XML parsing is a huge subject. The reference material is in section 20, *Structured Markup Processing Tools*, of the *Python Standard Library*. There are many modules to parse XML, each with different advantages and disadvantages. We'll focus on `ElementTree`.

Beyond these simple serialization formats, we can also have hybrid problems. One example of a hybrid problem is a spreadsheet encoded in XML. This means that we have a row-and-column data representation problem wrapped in the XML parsing problem. This leads to more complex software for disentangling the various kinds of data that were flattened to CSV-like rows so that we can recover useful Python objects.

In this chapter, we will cover the following topics:

- Understanding persistence class, state, and representation

- Filesystem and network considerations
- Defining classes to support persistence
- Dumping and loading with JSON
- Dumping and loading with YAML
- Dumping and loading with `pickle`
- Dumping and loading with CSV
- Dumping and loading with XML

Technical requirements

The code files for this chapter are available at `https://git.io/fj2Uw`.

Understanding persistence, class, state, and representation

Primarily, our Python objects exist in volatile computer memory. The upper bound on the life of an object is the duration of the Python process. This lifetime is further constrained by objects only lasting as long there are references to them. If we want an object with a longer duration, we need to make it persistent. If we want to extract the state of an object from one process and provide this state information to another process, the same serialization techniques for persistence can be used for the transfer of object state.

Most operating systems offer persistent storage in the form of a filesystem. This can include disk drives, flash drives, or other forms of non-volatile storage. Persisting the bytes from memory to the filesystem turns out to be surprisingly difficult.

The complexity arises because our in-memory Python objects have references to other objects. An object refers to its class. The class refers to any base classes. The object might be a container and refer to other objects. The in-memory version of an object is a web of references and relationships. The references are generally based on the locations in memory, which are not fixed: the relationships would be broken by trying to simply dump and restore memory bytes.

The web of references surrounding an object contains other objects that are largely static. Class definitions, for example, change very slowly compared to instance variables within an object. Python gives us a formal distinction between the instance variables of an object and other methods defined in the class. Consequently, serialization techniques focus on persisting the dynamic state of an object based on its instance variables.

We don't actually have to do anything extra to persist class definitions; we already have a very simple method for handling classes. Class definitions exist primarily as source code. The class definition in the volatile memory is rebuilt

from the source (or the byte-code version of the source) every time it's needed. If we need to exchange a class definition, we exchange Python modules or packages.

Let's take a look at common Python terminology in the next section.

Common Python terminology

Python serialization terminology tends to focus on the words *dump* and *load*. Most of the classes we're going to work with will define methods such as the following:

- `dump(object, file)`: This will dump the given object to a file.
- `dumps(object)`: This will dump an object, returning a string representation.
- `load(file)`: This will load an object from a file, returning the constructed object.
- `loads(string)`: This will load an object from a string representation, returning the constructed object.

There's no formal standard; the method names aren't *guaranteed* by any formal ABC inheritance or mixin class definition. However, they're widely used. Generally, the file used for the dump or load can be any *file-like* object.

To be useful, the file-like object must implement a short list of methods. Generally, `read()` and `readline()` are required for the load. We can, therefore, use the `io.StringIO` objects as well as the `urllib.request` objects as sources for the load. Similarly, dump places few requirements on the data source, mostly a `write()` method is used. We'll dig into these file object considerations next.

Filesystem and network considerations

As the OS filesystem (and network) works in bytes, we need to represent the values of an object's instance variables as a serialized stream of bytes. Often, we'll use a two-step transformation to get the bytes: firstly, we'll represent the state of an object as a string; secondly, we'll rely on the Python `str` class to provide bytes in a standard encoding. Python's built-in features for encoding a string into bytes neatly solves the second part of the problem. This allows most serialization methods to focus on creating strings.

When we look at our OS filesystems, we see two broad classes of devices: *block-mode* devices and *character-mode* devices. Block-mode devices can also be called *seekable* because the OS supports a seek operation that can access any byte in the file in an arbitrary order. Character-mode devices are not seekable; they are interfaces where bytes are transmitted serially. Seeking would involve some kind of time travel to recover past bytes or see future bytes.

This distinction between the *character* and *block* mode can have an impact on how we represent the state of a complex object or a collection of objects. The serializations we'll look at in this chapter focus on the simplest common feature set: an ordered stream of bytes. The stream of bytes can be written to either kind of device.

The formats we'll look at in `Chapter 11`, *Storing and Retrieving Objects via Shelve*, and `Chapter 12`, *Storing and Retrieving objects via SQLite*, however, will require block-mode storage in order to encode more objects than could possibly fit into memory. The `shelve` module and the `SQLite` database require seekable files on block-mode devices.

To an extent, the OS unifies block- and character-mode devices into a single filesystem metaphor. Some parts of the Python Standard Library implement the common feature set between the block and character devices. When we use Python's `urllib.request`, we can access the network resources as well as local files.

When we open a local file, the `urllib.request.urlopen()` function imposes the limited character-mode interface on an otherwise seekable file on a block-mode device. Because the distinction is invisible, it lets a single application work with network or local resources.

Let's define classes to support persistence.

Defining classes to support persistence

In order to work with persistence, we need some objects that we want to save. We'll look at a simple microblog and the posts on that blog. Here's a class definition for `Post`:

```
from dataclasses import dataclass
import datetime

@dataclass
class Post:
    date: datetime.datetime
    title: str
    rst_text: str
    tags: List[str]

    def as_dict(self) -> Dict[str, Any]:
        return dict(
            date=str(self.date),
            title=self.title,
            underline="-" * len(self.title),
            rst_text=self.rst_text,
            tag_text=" ".join(self.tags),
        )
```

The instance variables are the attributes of each microblog post – a date, a title, some text, and some tags. Our attribute name provides us with a hint that the text should be in **reStructuredText(reST)** markup, even though that's largely irrelevant to the rest of the data model.

To support simple substitution into templates, the `as_dict()` method returns a dictionary of values that have been converted into string format. We'll look at the processing templates using `string.Template` later. The type hint reflects the general nature of JSON, where the resulting object will be a dictionary with string keys and values chosen from a small domain of types. In this example, all of the values are strings, and `Dict[str, str]` could also be used; this seems overly specific, however, so `Dict[str, Any]` is used to provide future flexibility.

In addition to the essential data values, we've added a few values to help with creating the reST output. The `tag_text` attribute is a flattened text version of the tuple of tag values. The `underline` attribute produces an underline string with a

length that matches the title string; this helps the reST formatting work out nicely.

We'll also create a blog as a collection of posts. We'll make this collection more than a simple list by including an additional attribute of a title for the collection of posts. We have three choices for the collection design: wrap, extend, or invent a new class. We'll head off some confusion by providing this warning: don't extend a `list` if you intend to make it persistent.

> ***Extending an iterable object can be confusing***
> *When we extend a sequence class, we might get confused with some serialization algorithms. This may wind up bypassing the extended features we put in a subclass of a sequence. Wrapping a sequence is usually a better idea than extending it.*

This encourages us to look at wrapping or inventing. Since the blog posts form a simple sequence, there will be few places for confusion, and we can extend a list. Here's a collection of microblog posts. We've built in a list to create the `Blog` class:

```
from collections import defaultdict

class Blog_x(list):

    def __init__(self, title: str, posts: Optional[List[Post]]=None) -> None:
        self.title = title
        super().__init__(posts if posts is not None else [])

    def by_tag(self) -> DefaultDict[str, List[Dict[str, Any]]]:
        tag_index: DefaultDict[str, List[Dict[str, Any]]] = defaultdict(list)
        for post in self:
            for tag in post.tags:
                tag_index[tag].append(post.as_dict())
        return tag_index

    def as_dict(self) -> Dict[str, Any]:
        return dict(
            title=self.title,
            entries=[p.as_dict() for p in self]
        )
```

In addition to extending the `list` class, we've also included an attribute that is the title of the microblog. The initializer uses a common technique to avoid providing a mutable object as a default value. We've provided `None` as the default value for `posts`. If `posts` is `None`, we use a freshly-minted empty list, `[]`. Otherwise, we use the given value for posts.

Additionally, we've defined a method that indexes the posts by their tags. In the resulting `defaultdict`, each key is a tag's text. Each value is a list of posts that

shares the given tag.

To simplify the use of `string.Template`, we've added another `as_dict()` method that boils the entire blog down to a simple dictionary of strings and dictionaries. The idea here is to produce only built-in types that have simple string representations. In this case, the `Dict[str, Any]` type hint reflects a general approach the return value. Practically, the tile is a `str`, and the entries are a `List[Dict[str, Any]]`, based on the definition of the `Post` entries. The extra details don't seem to be completely helpful, so we've left the hint as `Dict[str, Any]`.

In the next section, we'll see how to render blogs and posts.

Rendering blogs and posts

We'll show you the template-rendering process next. In order to see how the rendering works, here's some sample data:

```
travel_x = Blog_x("Travel")
travel_x.append(
    Post(
        date=datetime.datetime(2013, 11, 14, 17, 25),
        title="Hard Aground",
        rst_text="""Some embarrassing revelation. Including ☹ and ⚓""",
        tags=["#RedRanger", "#Whitby42", "#ICW"],
    )
)
travel_x.append(
    Post(
        date=datetime.datetime(2013, 11, 18, 15, 30),
        title="Anchor Follies",
        rst_text="""Some witty epigram. Including < & > characters.""",
        tags=["#RedRanger", "#Whitby42", "#Mistakes"],
    )
)
```

We've serialized the `Blog` and `Post` objects in the form of Python code. This can be a nice way to represent the blog. There are some use cases where Python code is a perfectly fine representation for an object. In `Chapter 14`, *Configuration Files and Persistence*, we'll look more closely at using Python to encode data.

Here's one way to render the blog into reST; a similar approach can be used to create **Markdown** (**MD**). From this output file, the docutils `rst2html.py` tool can transform the reST output into the final HTML file. This saves us from having to digress into HTML and CSS. Also, we're going to use reST to write the documentation in `Chapter 20`, *Quality and Documentation*. For more information on docutils, see `Chapter 1`, *Preliminaries, Tools, and Techniques*.

We can use the `string.Template` class to do this. However, it's clunky and complex. There are a number of add-on template tools that can perform a more sophisticated substitution, including loops and conditional processing within the template itself. You can find a list of alternatives at `https://wiki.python.org/moin/Templating`. We're going to show you an example using the Jinja2 template tool (`https://pypi.python.org/pypi/Jinja2`). Here's a script to render this data in reST using a template:

```
from jinja2 import Template
blog_template= Template( """
{{title}}
{{underline}}

{% for e in entries %}
{{e.title}}
{{e.underline}}

{{e.rst_text}}

:date: {{e.date}}

:tags: {{e.tag_text}}
{% endfor %}

Tag Index
=========
{% for t in tags %}

*   {{t}}
    {% for post in tags[t] %}

    -   `{{post.title}}`_
    {% endfor %}
{% endfor %}
""")
print(blog_template.render(tags=travel.by_tag(), **travel.as_dict()))
```

The `{{title}}` and `{{underline}}` elements (and all similar elements) show us how values are substituted into the text of the template. The `render()` method is called with `**travel.as_dict()` to ensure that attributes such as `title` and `underline` will be keyword arguments.

The `{%for%}` and `{%endfor%}` constructs show us how Jinja can iterate through the sequence of `Post` entries in `Blog`. Within the body of this loop, the `e` variable will be the dictionary created from each `Post`. We've picked specific keys out of the dictionary for each post, such as `{{e.title}}` and `{{e.rst_text}}`.

We also iterated through a `tags` collection for the `Blog`. This is a dictionary with the keys of each tag and the posts for the tag. The loop will visit each key, assigned to `t`. The body of the loop will iterate through the posts in the dictionary value, `tags[t]`.

The ``{{post.title}}`_` construct uses reST markup to generate a link to the section that has that title within the document. This kind of very simple markup is one of the strengths of reST. It lets us use the blog titles as sections and links within the index. This means that the titles *must* be unique or we'll get reST rendering errors.

Because this template iterates through a given blog, it will render all of the posts in one smooth motion. The `string.Template`, which is built-in for Python, can't iterate. This makes it a bit more complex to render all of the `Posts` of a `Blog`.

Let's see how to dump and load using JSON.

Dumping and loading with JSON

What is JSON? A section from the `https://www.json.org/` web page states the following:

"JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language."

This format is used by a broad spectrum of languages and frameworks. Databases such as CouchDB represent their data as JSON objects, simplifying the transmission of data between applications. JSON documents have the advantage of looking vaguely like Python `list` and `dict` literal values. They're easy to read and easy to edit manually.

The `json` module works with the built-in Python types. It does not work with classes defined by us until we take some additional steps. We'll look at these extension techniques next. For the following Python types, there's a mapping to JavaScript types that JSON uses:

Python type	JSON
`dict`	`object`
`list`, `tuple`	`array`
`str`	`string`
`int`, `float`	`number`

`True`	`true`
`False`	`false`
`None`	`null`

Other types are not supported; this means an values of another type must be coerced to one of these types. This is often done via the extension functions that we can plug into the `dump()` and `load()` functions. We can explore these built-in types by transforming our microblog objects into simpler Python `lists` and `dicts`. When we look at our `Post` and `Blog` class definitions, we have already defined the `as_dict()` methods that reduce our custom class objects to built-in Python objects.

Here's the code required to produce a JSON version of our blog data:

```
import json
print(json.dumps(travel.as_dict(), indent=4))
```

Here's the output:

```
{
    "entries": [
        {
            "title": "Hard Aground",
            "underline": "------------",
            "tag_text": "#RedRanger #Whitby42 #ICW",
            "rst_text": "Some embarrassing revelation. Including \u2639 and \u2693",
            "date": "2013-11-14 17:25:00"
        },
        {
            "title": "Anchor Follies",
            "underline": "--------------",
            "tag_text": "#RedRanger #Whitby42 #Mistakes",
            "rst_text": "Some witty epigram. Including < & > characters.",
            "date": "2013-11-18 15:30:00"
        }
    ],
    "title": "Travel"
}
```

The preceding output shows us how each of the various objects are translated from Python to the JSON notation. What's elegant about this is that our Python objects have been written into a standardized notation. We can share them with

other applications. We can write them to disk files and preserve them. There are several unpleasant features of the JSON representation:

- We had to rewrite our Python objects into dictionaries. It would be much nicer to transform Python objects more simply, without explicitly creating additional dictionaries.
- We can't rebuild our original `Blog` and `Post` objects easily when we load this JSON representation. When we use `json.load()`, we won't get `Blog` or `Post` objects; we'll just get `dict` and list objects. We need to provide some additional hints to rebuild the `Blog` and `Post` objects.
- There are some values in the object's `__dict__` that we'd rather not persist, such as the underlined text for a `Post`.

We need something more sophisticated than the built-in JSON encoding.

Let's take a look at JSON type hints in the next section.

JSON type hints

The `as_dict()` methods in both the `Blog` class and `Post` class shown earlier use a simplistic `Dict[str, Any]` type hint for a data structure that's compatible with JSON serialization. While the type hint was meaningful for those examples, this isn't an ideal general description of the types used by JSON serialization.

The actual type that can be readily serialized *could* be defined like this:

```
from typing import Union, Dict, List, Type
JSON = Union[Dict[str, 'JSON'], List['JSON'], int, str, float, bool, Type[None]]
```

Currently, **mypy** doesn't handle recursive types gracefully. Therefore, we're forced to use the following:

```
JSON = Union[Dict[str, Any], List[Any], int, str, float, bool, Type[None]]
```

The classes defined in this chapter don't use this more general `JSON` type hint. In this chapter, we're only interested in working with the `Dict[str, Any]` subset of JSON-compatible Python objects. It can be helpful in some contexts to include the more sophisticated hint to be sure the types involved are correct.

Let's see how to support JSON in our classes.

Supporting JSON in our classes

In order to properly support creating strings in JSON notation, we need encoders and decoders for classes outside the types that can be converted automatically. To encode our unique objects into JSON, we need to provide a function that will reduce our objects to Python primitive types. The `json` module calls this a *default* function; it provides a default encoding for an object of an unknown class.

To decode strings in JSON notation and create Python objects of an application class, a class outside the baseline types supported by JSON, we need to provide an extra function. This extra function will transform a dictionary of Python primitive values into an instance of the one of our application classes. This is called the *object hook* function; it's used to transform `dict` into an object of a customized class.

The `json` module documentation suggests that we might want to make use of class hinting. The Python documentation includes a reference to the JSON-RPC version 1 specification (see `http://json-rpc.org/wiki/specification`). Their suggestion is to encode an instance of a customized class as a dictionary, like the following:

```
{"__jsonclass__": ["ClassName", [param1,...]]}
```

The suggested value associated with the `"__jsonclass__"` key is a list of two items: the class name, and a list of arguments required to create an instance of that class. The specification allows for more features, but they're not relevant to Python.

To decode an object from a JSON dictionary, an object hook function can look for the `"__jsonclass__"` key as a hint that one of our classes needs to be built, not a built-in Python object. The class name can be mapped to a class object and the argument sequence can be used to build the instance.

When we look at other sophisticated JSON encoders (such as the one that comes with the Django Web framework), we can see that they provide a bit more complex an encoding of a custom class. They include the class, a database primary key, and the attribute values. We'll look at how to implement customized

encoding and decoding. The rules are represented as simple functions that are plugged into the JSON encoding and decoding functions.

Let's see how to customize JSON encoding.

Customizing JSON encoding

For class hinting, we'll provide three pieces of information. We'll include a `__class__` key that names the target class. The `__args__` key will provide a sequence of positional argument values. A `__kw__` key will provide a dictionary of keyword argument values. (We will not use the `__jsonclass__` key; it's too long and doesn't seem Pythonic.) This will cover all the options of `__init__()`.

Here's an encoder that follows this design:

```
def blog_encode(object: Any) -> Dict[str, Any]:
    if isinstance(object, datetime.datetime):
        return dict(
            __class__="datetime.datetime",
            __args__=[],
            __kw__=dict(
                year=object.year,
                month=object.month,
                day=object.day,
                hour=object.hour,
                minute=object.minute,
                second=object.second,
            ),
        )
    elif isinstance(object, Post):
        return dict(
            __class__="Post",
            __args__=[],
            __kw__=dict(
                date=object.date,
                title=object.title,
                rst_text=object.rst_text,
                tags=object.tags,
            ),
        )
    elif isinstance(object, Blog):
        return dict(
            __class__="Blog", __args__=[object.title, object.entries], __kw__={}
        )
    else:
        return object
```

This function shows us two different flavors of object encodings for the three classes:

- We encoded a `datetime.datetime` object as a dictionary of individual fields using keyword arguments.
- We encoded a `Post` instance as a dictionary of individual fields, also using keyword arguments.

- We encoded a `Blog` instance as a sequence of title and post entries using a sequence of positional arguments.

The `else:` clause is used for all other classes: this invokes the existing encoder's default encoding. This will handle the built-in classes. We can use this function to encode as follows:

```
text = json.dumps(travel, indent=4, default=blog_encode)
```

We provided our function, `blog_encode()`, as the `default=` keyword parameter to the `json.dumps()` function. This function is used by the JSON encoder to determine the encoding for an object. This encoder leads to JSON objects that look like the following:

```
{
    "__args__": [
        "Travel",
        [
            {
                "__args__": [],
                "__kw__": {
                    "tags": [
                        "#RedRanger",
                        "#Whitby42",
                        "#ICW"
                    ],
                    "rst_text": "Some embarrassing revelation. Including \u2639 and \u26
                    "date": {
                        "__args__": [],
                        "__kw__": {
                            "minute": 25,
                            "hour": 17,
                            "day": 14,
                            "month": 11,
                            "year": 2013,
                            "second": 0
                        },
                        "__class__": "datetime.datetime"
                    },
                    "title": "Hard Aground"
                },
                "__class__": "Post"
            },
.
.
.
    "__kw__": {},
    "__class__": "Blog"
}
```

We've taken out the second blog entry because the output was rather long. A `Blog` object is now wrapped with a `dict` that provides the class and two positional argument values. The `Post` and `datetime` objects, similarly, are wrapped with the

class name and the keyword argument values.

Let's see how to customize JSON decoding.

Customizing JSON decoding

In order to decode objects from a string in JSON notation, we need to work within the structure of a JSON parsing. Objects of our customized class definitions were encoded as simple `dicts`. This means that each `dict` decoded by the JSON decoder *could* be one of our customized classes. Or, what is serialized as a `dict` should be left as a `dict`.

The JSON decoder *object hook* is a function that's invoked for each `dict` to see whether it represents a customized object. If `dict` isn't recognized by the `hook` function, it's an ordinary dictionary and should be returned without modification. Here's our object hook function:

```
def blog_decode(some_dict: Dict[str, Any]) -> Dict[str, Any]:
    if set(some_dict.keys()) == {"__class__", "__args__", "__kw__"}:
        class_ = eval(some_dict["__class__"])
        return class_(*some_dict["__args__"], **some_dict["__kw__"])
    else:
        return some_dict
```

Each time this function is invoked, it checks for the keys that define an encoding of our objects. If the three keys are present, the given function is called with the arguments and keywords. We can use this object hook to parse a JSON object, as follows:

```
blog_data = json.loads(text, object_hook=blog_decode)
```

This will decode a block of text, encoded in a JSON notation, using our `blog_decode()` function to transform `dict` into proper `Blog` and `Post` objects.

In the next section, we'll take a look at security and the `eval()` issue.

Security and the eval() issue

Some programmers will object to the use of the `eval()` function in our preceding `blog_decode()` function, claiming that it is a pervasive security problem. What's silly is the claim that `eval()` is a *pervasive* problem. It's a *potential* security problem if malicious code is written into the JSON representation of an object by some malicious actor. A local malicious actor has access to the Python source. Why waste their time on subtly tweaking JSON files? Why not just edit the Python source?

As a practical issue, we have to look at the transmission of the JSON documents through the internet; this is an actual security problem. However, even this problem does not indict `eval()` in general.

Some provisions can be made for a situation where an untrustworthy document has been tweaked by a **Man-in-the-Middle** (**MITM**) attack. Assume a JSON document is doctored while passing through a web interface that includes an untrustworthy server acting as a proxy. (SSL is the preferred method to prevent this problem, so we have to assume parts of the connection are also unsecured.)

If necessary, to cope with possible MITM attacks, we can replace `eval()` with a dictionary that maps from name to class. We can change the `class_ =eval(some_dict['__class__'])` expression to the following:

```
class_ = {
    "Post": Post,
    "Blog": Blog,
    "datetime.datetime": datetime.datetime
}[some_dict['__class__']]
```

This will prevent problems in the event that a JSON document is passed through a non-SSL-encoded connection. It also leads to a maintenance requirement to tweak this mapping each time the application design changes to introduce new classes.

Let's see how to refactor the `encode` function in the next section.

Refactoring the encode function

The encoding function shouldn't expose information on the classes being converted into JSON. To keep each class properly encapsulated, it seems better to refactor the creation of a serialized representation into each application class. We'd rather not pile all of the encoding rules into a function outside the class definitions.

To do this with library classes, such as `datetime`, we would need to extend `datetime.datetime` for our application. This leads to making our application use the extended `datetime` instead of the `datetime` library. This can become a bit of a headache to avoid using the built-in `datetime` classes. Consequently, we may elect to strike a balance between our customized classes and library classes. Here are two class extensions that will create JSON-encodable class definitions. We can add a property to `Blog`:

```
@property
def _json( self ) -> Dict[str, Any]:
    return dict(
        __class__=self.__class__.__name__,
        __kw__={},
        __args__=[self.title, self.entries]
    )
```

This property will provide initialization arguments that are usable by our decoding function. We can add this property to `Post`:

```
@property
def _json(self) -> Dict[str, Any]:
    return dict(
        __class__=self.__class__.__name__,
        __kw__=dict(
            date= self.date,
            title= self.title,
            rst_text= self.rst_text,
            tags= self.tags,
        ),
        __args__=[]
    )
```

As with `Blog`, this property will provide initialization arguments that are usable by our decoding function. The type hint emphasizes the intermediate, JSON-friendly representation of Python objects as `Dict[str, Any]`.

These two properties let us modify the encoder to make it somewhat simpler. Here's a revised version of the default function provided for encoding:

```
def blog_encode_2(object: Union[Blog, Post, Any) -> Dict[str, Any]:
    if isinstance(object, datetime.datetime):
        return dict(
            __class__="datetime.datetime",
            __args__=[],
            __kw__=dict(
                year= object.year,
                month= object.month,
                day= object.day,
                hour= object.hour,
                minute= object.minute,
                second= object.second,
            )
        )
    else:
        try:
            encoding = object._json
        except AttributeError:
            encoding = json.JSONEncoder().default(o)
        return encoding
```

There are two kinds of cases here. For a `datetime.datetime` library class, this function includes the serialization, exposing details of the implementation. For our `Blog` and `Post` application classes, we can rely on these classes having a consistent `_json()` method that emits a representation suitable for encoding.

Let's see how to standardize a `date` string in the next section.

Standardizing the date string

Our formatting of dates doesn't make use of the widely-used ISO standard text format. To be more compatible with other languages, we should properly encode the `datetime` object in a standard string and parse a standard string.

As we're already treating dates as a special case, this seems to be a sensible implementation. It can be done without too much change to our encoding and decoding. Consider this small change to the encoding:

```
if isinstance(object, datetime.datetime):
    fmt= "%Y-%m-%dT%H:%M:%S"
    return dict(
        __class__="datetime.datetime.strptime",
        __args__=[object.strftime(fmt), fmt],
        __kw__={}
    )
```

The encoded output names the static method `datetime.datetime.strptime()` and provides the argument-encoded `datetime` as well as the format to be used to decode it. The output for a post now looks like the following snippet:

```
{
    "__args__": [],
    "__class__": "Post_J",
    "__kw__": {
        "title": "Anchor Follies",
        "tags": [
            "#RedRanger",
            "#Whitby42",
            "#Mistakes"
        ],
        "rst_text": "Some witty epigram.",
        "date": {
            "__args__": [
                "2013-11-18T15:30:00",
                "%Y-%m-%dT%H:%M:%S"
            ],
            "__class__": "datetime.datetime.strptime",
            "__kw__": {}
        }
    }
}
```

This shows us that we now have an ISO-formatted date instead of individual fields. We've also moved away from the object creation using a class name. The `__class__` value is expanded to be a class name or a static method name.

Writing JSON to a file

When we write JSON files, we generally do something like this:

```
from pathlib import Path
with Path("temp.json").open("w", encoding="UTF-8") as target:
    json.dump(travel3, target, default=blog_j2_encode)
```

We open the file with the required encoding. We provide the file object to the `json.dump()` method. When we read JSON files, we will use a similar technique:

```
from pathlib import Path
with Path("some_source.json").open(encoding="UTF-8") as source:
     objects = json.load(source, object_hook=blog_decode)
```

The idea is to segregate the JSON representation as text from any conversion to bytes on the resulting file. There are a few formatting options that are available in JSON. We've shown you an indent of four spaces because that seems to produce nice-looking JSON. As an alternative, we can make the output more compact by leaving the indent option. We can compact it even further by making the separators more terse.

The following is the output created in `temp.json`:


```
{"__class__":"Blog_J","__args__":["Travel",[{"__class__":"Post_J","__args__":[],"__kw__"
```


Let's see how to dump and load using YAML.

Dumping and loading with YAML

The `https://yaml.org/` web page states the following about YAML:

YAML™ (rhymes with "camel") is a human-friendly, cross-language, Unicode-based data serialization language designed around the common native data types of agile programming languages.

The Python Standard Library documentation for the `json` module explains the following about JSON and YAML:

JSON is a subset of YAML 1.2. The JSON produced by this module's default settings (in particular, the default separators value) is also a subset of YAML 1.0 and 1.1. This module can thus also be used as a YAML serializer.

Technically, then, we can prepare YAML data using the `json` module. However, the `json` module cannot be used to de-serialize more sophisticated YAML data. There are two benefits to using YAML. First, it's a more sophisticated notation, allowing us to encode additional details about our objects. Second, the PyYAML implementation has a deep level of integration with Python that allows us to very simply create YAML encodings of Python objects. The drawback of YAML is that it is not as widely used as JSON. We'll need to download and install a YAML module. A good one can be found at `http://pyyaml.org/wiki/PyYAML`.

Once we've installed the package, we can dump our objects into the YAML notation:

```
import yaml
text = yaml.dump(travel2)
print(text)
```

Here's what the YAML encoding for our microblog looks like:


```
!!python/object:__main__.Blog
entries:
- !!python/object:__main__.Post
  date: 2013-11-14 17:25:00
  rst_text: Some embarrassing revelation. Including ☹ and ⚓
  tags: !!python/tuple ['#RedRanger', '#Whitby42', '#ICW']
  title: Hard Aground
- !!python/object:__main__.Post
  date: 2013-11-18 15:30:00
  rst_text: Some witty epigram. Including < & > characters.
  tags: !!python/tuple ['#RedRanger', '#Whitby42', '#Mistakes']
  title: Anchor Follies
```

The output is relatively terse but also delightfully complete. Also, we can easily edit the YAML file to make updates. The class names are encoded with a YAML `!!` tag. YAML contains 11 standard tags. The `yaml` module includes a dozen Python-specific tags, plus five *complex* Python tags.

The Python class names are qualified by the defining module. In our case, the module happened to be a simple script, so the class names are `__main__.Blog` and `__main__.Post`. If we had imported these from another module, the class names would reflect the module that defined the classes.

Items in a list are shown in a block sequence form. Each item starts with a `-` sequence; the rest of the items are indented with two spaces. When `list` or `tuple` is small enough, it can flow onto a single line. If it gets longer, it will wrap onto multiple lines. To load Python objects from a YAML document, we can use the following code:

```
copy = yaml.load(text)
```

This will use the tag information to locate the class definitions and provide the values found in the YAML document to the `constructor` class. Our microblog objects will be fully reconstructed.

In the next chapter, we'll format YAML data on a file.

Formatting YAML data on a file

When we write YAML files, we generally do something like this:

```
from pathlib import Path
import yaml
with Path("some_destination.yaml").open("w", encoding="UTF-8") as target:
    yaml.dump(some_collection, target)
```

We open the file with the required encoding. We provide the file object to the `yaml.dump()` method; the output is written there. When we read YAML files, we will use a similar technique:

```
from pathlib import Path
import yaml
with Path("some_source.yaml").open(encoding="UTF-8") as source:
     objects= yaml.load(source)
```

The idea is to segregate the YAML representation as text from any conversion to bytes on the resulting file. We have several formatting options to create a prettier YAML representation of our data. Some of the options are shown in the following table:

`explicit_start`	If `true`, writes a `---` marker before each object.
`explicit_end`	If `true`, writes a `...` marker after each object. We might use this or `explicit_start` if we're dumping a sequence of YAML documents into a single file and need to know when one ends and the next begins.
`version`	Given a pair of integers (*x*, *y*), writes a `%YAML x.y` directive at the beginning. This should be `version=(1,2)`.
`tags`	Given a mapping, it emits a YAML `%TAG` directive with

	different tag abbreviations.
`canonical`	If `true`, includes a tag on every piece of data. If `false`, a number of `tags` are assumed.
`indent`	If set to a number, changes the indentation used for blocks.
`width`	If set to a number, changes the `width` at which long items are wrapped to multiple, indented lines.
`allow_unicode`	If set to `true`, permits full Unicode without escapes. Otherwise, characters outside the ASCII subset will have escapes applied.
`line_break`	Uses a different line-ending character; the default is a newline.

Of these options, `explicit_end` and `allow_unicode` are perhaps the most useful.

Extending the YAML representation

Sometimes, one of our classes has a tidy representation that is nicer than the default YAML dump of attribute values. For example, the default YAML for our Blackjack `Card` class definitions will include several derived values that we don't really need to preserve.

The `yaml` module includes a provision for adding a `representer` and `constructor` to a class definition. The `representer` is used to create a YAML representation, including a tag and value. The `constructor` is used to build a Python object from the given value. Here's yet another `Card` class hierarchy:

```
from enum import Enum
class Suit(str, Enum):
    Clubs = "♣"
    Diamonds = "♦"
    Hearts = "♥"
    Spades = "♠"

class Card:

    def __init__(self, rank: str, suit: Suit,
        hard: Optional[int]=None,
        soft: Optional[int]=None
    ) -> None:
        self.rank = rank
        self.suit = suit
        self.hard = hard or int(rank)
        self.soft = soft or int(rank)

    def __str__(self) -> str:
        return f"{self.rank!s}{self.suit.value!s}"

class AceCard(Card):

    def __init__(self, rank: str, suit: Suit) -> None:
        super().__init__(rank, suit, 1, 11)


class FaceCard(Card):

    def __init__(self, rank: str, suit: Suit) -> None:
        super().__init__(rank, suit, 10, 10)
```

We've used the superclass, `Card`, for number cards and defined two subclasses, `AceCard` and `FaceCard`, for aces and face cards. In previous examples, we made extensive use of a factory function to simplify the construction. The factory handled mapping from a rank of `1` to a class of `AceCard`, and from ranks of `11`, `12`,

and `13` to a class of `FaceCard`. This was essential so that we could easily build a deck using a simple `range(1,14)` for the rank values.

When loading from YAML, the class will be fully spelled out via the YAML `!!` tags. The only missing information would be the hard and soft values associated with each subclass of the card. The hard and soft points have three relatively simple cases that can be handled through optional initialization parameters. Here's how it looks when we dump these objects into the YAML format using default serialization:

```
- !!python/object:Chapter_10.ch10_ex2.AceCard
  hard: 1
  rank: A
  soft: 11
  suit: !!python/object/apply:Chapter_10.ch10_ex2.Suit
  - ♣
- !!python/object:Chapter_10.ch10_ex2.Card
  hard: 2
  rank: '2'
  soft: 2
  suit: !!python/object/apply:Chapter_10.ch10_ex2.Suit
  - ♥
- !!python/object:Chapter_10.ch10_ex2.FaceCard
  hard: 10
  rank: K
  soft: 10
  suit: !!python/object/apply:Chapter_10.ch10_ex2.Suit
  - ♦
```

These are correct, but perhaps a bit wordy for something as simple as a playing card. We can extend the `yaml` module to produce smaller and more focused output for these simple objects. Let's define `representer` and `constructor` for our `Card` subclasses. Here are the three functions and registrations:

```
def card_representer(dumper: Any, card: Card) -> str:
    return dumper.represent_scalar(
        "!Card", f"{card.rank!s}{card.suit.value!s}")


def acecard_representer(dumper: Any, card: Card) -> str:
    return dumper.represent_scalar(
        "!AceCard", f"{card.rank!s}{card.suit.value!s}")


def facecard_representer(dumper: Any, card: Card) -> str:
    return dumper.represent_scalar(
        "!FaceCard", f"{card.rank!s}{card.suit.value!s}")

yaml.add_representer(Card, card_representer)
yaml.add_representer(AceCard, acecard_representer)
yaml.add_representer(FaceCard, facecard_representer)
```

We've represented each `Card` instance as a short string. YAML includes a tag to show which class should be built from the string. All three classes use the same format string. This happens to match the `__str__()` method, leading to a potential optimization.

The other problem we need to solve is constructing `Card` instances from the parsed YAML document. For that, we need constructors. Here are three constructors and the registrations:

```
def card_constructor(loader: Any, node: Any) -> Card:
    value = loader.construct_scalar(node)
    rank, suit = value[:-1], value[-1]
    return Card(rank, suit)


def acecard_constructor(loader: Any, node: Any) -> Card:
    value = loader.construct_scalar(node)
    rank, suit = value[:-1], value[-1]
    return AceCard(rank, suit)


def facecard_constructor(loader: Any, node: Any) -> Card:
    value = loader.construct_scalar(node)
    rank, suit = value[:-1], value[-1]
    return FaceCard(rank, suit)

yaml.add_constructor("!Card", card_constructor)
yaml.add_constructor("!AceCard", acecard_constructor)
yaml.add_constructor("!FaceCard", facecard_constructor)
```

As a scalar value is parsed, the tag will be used to locate a specific `constructor`. The `constructor` can then decompose the string and build the proper subclass of a `Card` instance. Here's a quick demo that dumps one card of each class:

```
deck = [AceCard("A", Suit.Clubs), Card("2", Suit.Hearts), FaceCard("K", Suit.Diamonds)]
text = yaml.dump(deck, allow_unicode=True)
```

The following is the output:

```
- !AceCard 'A♣'
- !Card '2♥'
- !FaceCard 'K♦'
```

This gives us short, elegant YAML representations of cards that can be used to reconstruct Python objects.

We can rebuild our three-card deck using the following statement:

```
yaml.load(text, Loader=yaml.Loader)
```

This will parse the representation, use the `constructor` functions, and build the expected objects. Because the `constructor` function ensures that proper initialization gets done, the internal attributes for the hard and soft values are properly rebuilt.

It's essential to use a specific `Loader` when adding new constructors to the `yaml` module. The default behavior is to ignore these additional `constructor` tags. When we want to use them, we need to provide a `Loader` that will handle extension tags.

Let's take a look at security and safe loading in the next section.

Security and safe loading

In principle, YAML can build objects of any type. This allows an attack on an application that transmits YAML files over the internet without proper SSL controls in place.

The YAML module offers a `safe_load()` method that refuses to execute arbitrary Python code as part of building an object. This severely limits what can be loaded. For insecure data exchanges, we can use `yaml.safe_load()` to create Python `dict` and `list` objects that contain only built-in types. We can then build our application classes from the `dict` and `list` instances. This is vaguely similar to the way we use JSON or CSV to exchange `dict` that must be used to create a proper object.

A better approach is to use the `yaml.YAMLObject` mixin class for our own objects. We use this to set some class-level attributes that provide hints to `yaml` and ensure the safe construction of objects.

Here's how we define a superclass for safe transmission:

```
class Card2(yaml.YAMLObject):
    yaml_tag = '!Card2'
    yaml_loader = yaml.SafeLoader
```

The two attributes will alert `yaml` that these objects can be safely loaded without executing arbitrary and unexpected Python code. Each subclass of `Card2` only has to set the unique YAML tag that will be used:

```
class AceCard2(Card2):
    yaml_tag = '!AceCard2'
```

We've added an attribute that alerts `yaml` that these objects use only this class definition. The objects can be safely loaded; they don't execute arbitrary untrustworthy code.

With these modifications to the class definitions, we can now use `yaml.safe_load()` on the YAML stream without worrying about the document having malicious code inserted over an unsecured internet connection. The explicit use of the

`yaml.YAMLObject` mixin class for our own objects coupled with setting the `yaml_tag` attribute has several advantages. It leads to slightly more compact files. It also leads to a better-looking YAML files—the long, generic `!!python/object:Chapter_10.ch10_ex2.AceCard` tags are replaced with shorter `!AceCard2` tags.

Let's see how to dump and load using pickle.

Dumping and loading with pickle

The `pickle` module is Python's native format to make objects persistent. The Python Standard Library (`https://docs.python.org/3/library/pickle.html`) says this about `pickle`:

The pickle module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure. Perhaps the most obvious thing to do with these byte streams is to write them onto a file, but it is also conceivable to send them across a network or store them in a database.

The focus of `pickle` is Python, and only Python. This is not a data-interchange format, such as JSON, YAML, CSV, or XML, that can be used with applications written in other languages.

The `pickle` module is tightly integrated with Python in a variety of ways. For example, the `__reduce__()` and `__reduce_ex__()` methods of a class exist to support the `pickle` processing.

We can easily pickle our microblog in the following manner:

```
import pickle
from pathlib import Path
with Path("travel_blog.p").open("wb") as target:
    pickle.dump(travel, target)
```

This exports the entire `travel` object to the given file. The file is written as raw bytes, so the `open()` function uses the `"wb"` mode.

We can easily recover a picked object in the following manner:

```
import pickle
from pathlib import Path
with Path("travel_blog.p").open("rb") as source:
    copy = pickle.load(source)
```

As pickled data is written as bytes, the file must be opened in the `"rb"` mode. The pickled objects will be correctly bound to the proper class definitions. The underlying stream of bytes is not intended for human consumption. It is readable after a fashion, but it is not designed for readability as YAML is.

We'll design a class for reliable pickle processing in the next section.

Designing a class for reliable pickle processing

The `__init__()` method of a class is not actually used to unpickle an object. The `__init__()` method is bypassed by using `__new__()` and setting the pickled values into the object's `__dict__` directly. This distinction matters when our class definition includes some processing in `__init__()`. For example, if `__init__()` opens external files, creates some part of a GUI, or performs some external update to a database, then this will not be performed during unpickling.

If we compute a new instance variable during the `__init__()` processing, there is no real problem. For example, consider a Blackjack `Hand` object that computes the total of the `Card` instances when the `Hand` is created. The ordinary `pickle` processing will preserve this computed instance variable. It won't be recomputed when the object is unpickled. The previously computed value will simply be unpickled.

A class that relies on processing during `__init__()` has to make special arrangements to be sure that this initial processing will happen properly. There are two things we can do:

- Avoid eager startup processing in `__init__()`. Instead, do only the minimal initialization processing. For example, if there are external file operations, these should be deferred until required. If there are any eager summarization computations, they must be redesigned to be done lazily. Similarly, any initialization logging will not be executed properly.
- Define the `__getstate__()` and `__setstate__()` methods that can be used by `pickle` to preserve the state and restore the state. The `__setstate__()` method can then invoke the same method that `__init__()` invokes to perform a one-time initialization processing in ordinary Python code.

We'll look at an example where the initial `Card` instances loaded into a `Hand` are logged for audit purposes by the `__init__()` method. Here's a version of `Hand` that doesn't work properly when unpickling:

```
audit_log = logging.getLogger("audit")
```

```
class Hand_bad:

    def __init__(self, dealer_card: Card, *cards: Card) -> None:
        self.dealer_card = dealer_card
        self.cards = list(cards)
        for c in self.cards:
            audit_log.info("Initial %s", c)

    def append(self, card: Card) -> None:
        self.cards.append(card)
        audit_log.info("Hit %s", card)

    def __str__(self) -> str:
        cards = ", ".join(map(str, self.cards))
        return f"{self.dealer_card} | {cards}"
```

This has two logging locations: during `__init__()` and `append()`. The `__init__()` processing works nicely for most cases of creating a `Hand_bad` object. It doesn't work when unpickling to recreate a `Hand_bad` object. Here's the logging setup to see this problem:

```
import logging,sys
audit_log = logging.getLogger("audit")
logging.basicConfig(stream=sys.stderr, level=logging.INFO)
```

This setup creates the log and ensures that the logging level is appropriate for seeing the audit information. Here's a quick script that builds, pickles, and unpickles `Hand`:

```
>>> h = Hand_bad(FaceCard("K", "♦"), AceCard("A", "♣"), Card("9", "♥"))
INFO:audit:Initial A♣
INFO:audit:Initial 9♥
>>> data = pickle.dumps(h)
>>> h2 = pickle.loads(data)
```

When we execute this, we see that the log entries written during `__init__()` processing. These entries are not written when unpickling `Hand`. Any other `__init__()` processing would also be skipped.

In order to properly write an audit log for unpickling, we could put lazy logging tests throughout this class. For example, we could extend `__getattribute__()` to write the initial log entries whenever any attribute is requested from this class. This leads to stateful logging and an `if` statement that is executed every time a hand object does something. A better solution is to tap into the way state is saved and recovered by `pickle`.

```
class Hand2:

    def __init__(self, dealer_card: Card, *cards: Card) -> None:
```

```
        self.dealer_card = dealer_card
        self.cards = list(cards)
        for c in self.cards:
            audit_log.info("Initial %s", c)

    def append(self, card: Card) -> None:
        self.cards.append(card)
        audit_log.info("Hit %s", card)

    def __str__(self) -> str:
        cards = ", ".join(map(str, self.cards))
        return f"{self.dealer_card} | {cards}"

    def __getstate__(self) -> Dict[str, Any]:
        return vars(self)

    def __setstate__(self, state: Dict[str, Any]) -> None:
        # Not very secure -- hard for mypy to detect what's going on.
        self.__dict__.update(state)
        for c in self.cards:
            audit_log.info("Initial (unpickle) %s", c)
```

The `__getstate__()` method is used while picking to gather the current state of the object. This method can return anything. In the case of objects that have internal memoization caches, for example, the cache might not be pickled in order to save time and space. This implementation uses the internal `__dict__` without any modification.

The `__setstate__()` method is used while unpickling to reset the value of the object. This version merges the state into the internal `__dict__` and then writes the appropriate logging entries.

In the next section, we'll take a look at security and the global issue.

Security and the global issue

During unpickling, a global name in the pickle stream can lead to the evaluation of arbitrary code. Generally, the global names inserted into the bytes are class names or a function name. However, it's possible to include a global name that is a function in a module such as `os` or `subprocess`. This allows an attack on an application that attempts to transmit pickled objects through the internet without strong SSL controls in place. In order to prevent the execution of arbitrary code, we must extend the `pickle.Unpickler` class. We'll override the `find_class()` method to replace it with something more secure. We have to account for several unpickling issues, such as the following:

- We have to prevent the use of the built-in `exec()` and `eval()` functions.
- We have to prevent the use of modules and packages that might be considered unsafe. For example, `sys` and `os` should be prohibited.
- We have to permit the use of our application modules.

Here's an example that imposes some restrictions:

```
import builtins

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module: str, name: str) -> Any:
        if module == "builtins":
            if name not in ("exec", "eval"):
                return getattr(builtins, name)
        elif module in ("__main__", "Chapter_10.ch10_ex3", "ch10_ex3"):
            # Valid module names depends on execution context.
            return globals()[name]
        # elif module in any of our application modules...
        elif module in ("Chapter_10.ch10_ex2",):
            return globals()[name]
        raise pickle.UnpicklingError(
            f"global '{module}.{name}' is forbidden"
        )
```

This version of the `Unpickler` class will help us avoid a large number of potential problems that could stem from a pickle stream that was doctored. It permits the use of any built-in function except `exec()` and `eval()`. It permits the use of classes defined only in `__main__`. In all other cases, it raises an exception.

Let's see how to dump and load using CSV.

Dumping and loading with CSV

The `csv` module encodes and decodes simple `list` or `dict` instances into a CSV notation. As with the `json` module, discussed previously, this is not a very complete persistence solution. The wide adoption of CSV files, however, means that it often becomes necessary to convert between Python objects and CSV.

Working with CSV files involves a manual mapping between potentially complex Python objects and very simplistic CSV structures. We need to design the mapping carefully, remaining cognizant of the limitations of the CSV notation. This can be difficult because of the mismatch between the expressive powers of objects and the tabular structure of a CSV file.

The content of each column of a CSV file is, by definition, pure text. When loading data from a CSV file, we'll need to convert these values into more useful types inside our applications. This conversion can be complicated by the way spreadsheets perform unexpected type coercion. We might, for example, have a spreadsheet where US zip codes have been changed into floating-point numbers by the spreadsheet application. When the spreadsheet saves to CSV, the zip codes could become a confusing numeric value. Bangor, Maine, for example, has a zip code of 04401. This becomes 4401 when converted into a number by a spreadsheet program.

Consequently, we might need to use a conversion such as `row['zip'].zfill(5)` or `('00000'+row['zip'])[-5:]` to restore the leading zeroes. Also, don't forget that a file might have a mixture of ZIP and ZIP and four postal codes, making this data cleansing even more challenging.

To further complicate working with CSV files, we have to be aware that they're often touched manually and are become subtly incompatible because of human tweaks. It's important for software to be flexible in the face of real-world irregularities that arise.

When we have relatively simple class definitions, we can often transform each instance into a simple, flat row of data values. Often, `NamedTuple` is a good match between a CSV source file and Python objects. Going the other way, we might

need to design our Python classes around `NamedTuple` if our application will save data in the CSV notation.

When we have classes that are containers, we often have a difficult time determining how to represent structured containers in flat CSV rows. This is an **impedance mismatch** between object models and the flat normalized tabular structure used for CSV files or relational databases. There's no good solution for the impedance mismatch; it requires careful design. We'll start with simple, flat objects to show you some CSV mappings.

Let's see how to dump simple sequences into CSV.

Dumping simple sequences into CSV

An ideal mapping is between the `NamedTuple` instances and rows in a CSV file. Each row represents a different `NamedTuple`. Consider the following Python class definition:

```
from typing import NamedTuple

class GameStat(NamedTuple):
    player: str
    bet: str
    rounds: int
    final: float
```

We've defined the objects in this application to have a simple, flat sequence of attributes. The Database architects call this **First Normal Form**. There are no repeating groups and each item is an atomic piece of data.

We might produce these objects from a simulation that looks like the following code:

```
from typing import Iterator, Type

def gamestat_iter(
    player: Type[Player_Strategy], betting: Type[Betting], limit: int = 100
) -> Iterator[GameStat]:
    for sample in range(30):
        random.seed(sample)  # Assures a reproducible result
        b = Blackjack(player(), betting())
        b.until_broke_or_rounds(limit)
        yield GameStat(player.__name__, betting.__name__, b.rounds, b.betting.stake)
```

This iterator will create Blackjack simulations with a given player and betting strategy. It will execute the game until the player is broke or has sat at the table for 100 individual rounds of play. At the end of each session, it will yield a `GameStat` object with the player strategy, betting strategy, the number of rounds, and the final stake. This will allow us to compute statistics for each play, betting strategy, or combination. Here's how we can write this to a file for later analysis:

```
import csv
from pathlib import Path

with (Path.cwd() / "data" / "ch10_blackjack_1.csv").open("w", newline="") as target:
    writer = csv.DictWriter(target, GameStat._fields)
    writer.writeheader()
    for gamestat in gamestat_iter(Player_Strategy, Martingale_Bet):
```

```
        writer.writerow(gamestat._asdict())
```

There are three steps to create a CSV writer:

1. Open a file with the newline option set to `""`. This will support the (possibly) nonstandard line ending for CSV files.
2. Create a CSV writer. In this example, we created `DictWriter` because it allows us to easily create rows from dictionary objects. The `GameStat._fields` attribute provides the Python attribute names so the CSV columns will precisely match the attributes of the `GameStat` subclass of the `NamedTuple` class.
3. Write a header in the first line of the file. This makes data exchange slightly simpler by providing some hint as to what's in the CSV file.

Once the `writer` object has been prepared, we can use the writer's `writerow()` method to write each dictionary to the CSV file. We can, to an extent, simplify this slightly by using the `writerows()` method. This method expects an iterator instead of an individual row. Here's how we can use `writerows()` with an iterator:

```
data = gamestat_iter(Player_Strategy, Martingale_Bet)
with (Path.cwd() / "data" / "ch10_blackjack_2.csv").open("w", newline="") as target:
    writer = csv.DictWriter(target, GameStat._fields)
    writer.writeheader()
    writer.writerows(g._asdict() for g in data)
```

We've assigned the iterator to a variable, `data`. For the `writerows()` method, we get a dictionary from each row produced by the iterator.

Let's load a simple sequence from CSV.

Loading simple sequences from CSV

We can load simple, sequential objects from a CSV file with a loop like the following:

```
with (Path.cwd() / "data" / "ch10_blackjack_1.csv").open() as source:
    reader = csv.DictReader(source)
    assert set(reader.fieldnames) == set(GameStat._fields)
    for gs in (GameStat(**r) for r in reader):
        print( gs )
```

We've defined a reader for our file. As we know that our file has a proper heading, we can use `DictReader`. This will use the first row to define the attribute names. We can now construct the `GameStat` objects from the rows in the CSV file. We've used a generator expression to build rows.

In this case, we've assumed that the column names match the attribute names of our `GameStat` class definition. We can, if necessary, confirm that the file matches the expected format by comparing `reader.fieldnames` with `GameStat._fields`. As the order doesn't have to match, we need to transform each list of field names into a set. Here's how we can check the column names:

```
assert set(reader.fieldnames) == set(GameStat._fields)
```

We've ignored the data types of the values that were read from the file. The two numeric columns will wind up being string values when we read from the CSV file. Because of this, we need a more sophisticated row-by-row transformation to create proper data values.

Here's a typical factory function that performs the required conversions:

```
def gamestat_rdr_iter(
        source_data: Iterator[Dict[str, str]]
    ) -> Iterator[GameStat]:
    for row in source_data:
        yield GameStat(row["player"], row["bet"], int(row["rounds"]), int(row["final"]))
```

We've applied the `int` function to the columns that are supposed to have numeric values. In the rare event where the file has the proper headers but improper data, we'll get an ordinary `ValueError` from a failed `int()` function. We can use this generator function as follows:

```
with (Path.cwd()/"data"/"ch10_blackjack_1.csv").open() as source:
    reader = csv.DictReader(source)
    assert set(reader.fieldnames) == set(GameStat._fields)
    for gs in gamestat_rdr_iter(reader):
        print(gs)
```

This version of the reader has properly reconstructed the `GameStat` objects by performing conversions on the numeric values.

Let's see how to handle containers and complex classes.

Handling containers and complex classes

When we look back at our microblog example, we have a `Blog` object that contains many `Post` instances. We designed `Blog` as a wrapper around `list`, so that the `Blog` would contain a collection. When working with a CSV representation, we have to design a mapping from a complex structure to a tabular representation. We have three common solutions:

- We can create two files, a blog file and a posting file. The blog file has only the `Blog` instances. Each `Blog` has a title in our example. Each `Post` row can then have a reference to the `Blog` row that the posting belongs to. We need to add a key for each `Blog`. Each `Post` would then have a foreign key reference to the `Blog` key.
- We can create two kinds of rows in a single file. We will have the `Blog` rows and `Post` rows. Our writers entangle the various types of data; our readers must disentangle the types of data.
- We can perform a relational database join between the various kinds of rows, repeating the `Blog` parent information on each `Post` child.

There's no *best* solution among these choices. We have to design a solution to cope with an impedance mismatch between flat CSV rows and more structured Python objects. The use cases for the data will define some of the advantages and disadvantages.

Creating two files requires that we create some kind of unique identifier for each `Blog` so that a `Post` can properly refer to the `Blog`. We can't easily use the Python internal ID, as these are not guaranteed to be consistent each time Python runs.

A common assumption is that the `Blog` title is a unique key; as this is an attribute of `Blog`, it is called a natural primary key. This rarely works out well; we cannot change a `Blog` title without also updating all of the `Posts` that refer to the `Blog`. A better plan is to invent a unique identifier and update the class design to include that identifier. This is called a **surrogate key**. The Python `uuid` module can provide unique identifiers for this purpose.

The code to use multiple files is nearly identical to the previous examples. The only change is to add a proper primary key to the `Blog` class. Once we have the keys defined, we can create writers and readers as shown previously to process the `Blog` and `Post` instances into their separate files.

In the next section, we'll dump and load multiple row types into a CSV file.

Dumping and loading multiple row types into a CSV file

Creating multiple kinds of rows in a single file makes the format a bit more complex. The column titles must become a union of all the available column titles. Because of the possibility of name clashes between the various row types, we can either access rows by position (preventing us from simply using `csv.DictReader`) or we must invent a more sophisticated column title that combines class and attribute names.

The process is simpler if we provide each row with an extra column that acts as a class discriminator. This extra column shows us what type of object the row represents. The object's class name would work well for this. Here's how we might write blogs and posts to a single CSV file using two different row formats:

```
with (Path.cwd() / "data" / "ch10_blog3.csv").open("w", newline="") as target:
    wtr = csv.writer(target)
    wtr.writerow(["__class__", "title", "date", "title", "rst_text", "tags"])
    for b in blogs:
        wtr.writerow(["Blog", b.title, None, None, None, None])
        for p in b.entries:
            wtr.writerow(["Post", None, p.date, p.title, p.rst_text, p.tags])
```

We created two varieties of rows in the file. Some rows have `'Blog'` in the first column and contain just the attributes of a `Blog` object. Other rows have `'Post'` in the first column and contain just the attributes of a `Post` object.

We did not make the column titles unique, so we can't use dictionary writers or readers. When allocating columns by position like this, each row allocates unused columns based on the other types of rows it must coexist with. These additional columns are filled with `None`. As the number of distinct row types grows, keeping track of the various positional column assignments can become challenging.

Also, the individual data type conversions can be somewhat baffling. In particular, we've ignored the data type of the timestamp and tags. We can try to reassemble our `Blogs` and `Posts` by examining the row discriminators:

```
with (Path.cwd() / "data" / "ch10_blog3.csv").open() as source:
    rdr = csv.reader(source)
    header = next(rdr)
    assert header == ["__class__", "title", "date", "title", "rst_text", "tags"]
    blogs = []
    for r in rdr:
        if r[0] == "Blog":
            blog = Blog(*r[1:2])  # type: ignore
            blogs.append(blog)
        elif r[0] == "Post":
            post = Post(*r[2:])  # type: ignore
            blogs[-1].append(post)
```

This snippet will construct a list of `Blog` objects. Each `'Blog'` row uses columns in `slice(1,2)` to define the `Blog` object. Each `'Post'` row uses columns in `slice(2,6)` to define a `Post` object. This requires that each `Blog` be followed by the relevant `Post` instances. A foreign key is not used to tie the two objects together.

We've used two assumptions about the columns in the CSV file that has the same order and type as the parameters of the class constructors. For `Blog` objects, we used `blog = Blog(*r[1:2])` because the one-and-only column is text, which matches the `constructor` class. When working with externally-supplied data, this assumption might prove to be invalid.

The `# type: ignore` comments are required because the data types from the reader will be strings and those types don't match the dataclass type definitions provided above. Subverting **mypy** checks to construct objects isn't ideal.

To build the `Post` instances and perform the appropriate type conversion, a separate function is required. This function will map the types and invoke the `constructor` class. Here's a mapping function to build `Post` instances:

```
import ast

def post_builder(row: List[str]) -> Post:
    return Post(
        date=datetime.datetime.strptime(row[2], "%Y-%m-%d %H:%M:%S"),
        title=row[3],
        rst_text=row[4],
        tags=ast.literal_eval(row[5]),
    )
```

This will properly build a `Post` instance from a row of text. It converts the text for `datetime` and the text for the tags into their proper Python types. This has the advantage of making the mapping explicit.

In this example, we're using `ast.literal_eval()` to decode more complex Python

literal values. This allows the CSV data to include the literal representation of a tuple of string values: `"('#RedRanger', '#Whitby42', '#ICW')"`. Without using `ast.literal_eval()`, we'd have to write our own parser for the rather complex regular expression around this data type. Instead of writing our own parser, we elected to serialize a tuple-of-string object that could be deserialized securely.

Let's see how to filter CSV rows with an iterator.

Filtering CSV rows with an iterator

We can refactor the previous load example to iterate through the `Blog` objects rather than constructing a list of the `Blog` objects. This allows us to skim through a large CSV file and locate just the relevant `Blog` and `Post` rows. This function is a generator that yields each individual `Blog` instance separately:

```
def blog_iter(source: TextIO) -> Iterator[Blog]:
    rdr = csv.reader(source)
    header = next(rdr)
    assert header == ["__class__", "title", "date", "title", "rst_text", "tags"]
    blog = None
    for r in rdr:
        if r[0] == "Blog":
            if blog:
                yield blog
            blog = blog_builder(r)
        elif r[0] == "Post":
            post = post_builder(r)
            blog.append(post)
    if blog:
        yield blog
```

This `blog_iter()` function creates the `Blog` object and appends the `Post` objects. Each time a `Blog` header appears, the previous `Blog` is complete and can be yielded. At the end, the final `Blog` object must also be yielded. If we want the large list of `Blog` instances, we can use the following code:

```
with (Path.cwd()/"data"/"ch10_blog3.csv").open() as source:
    blogs = list(blog_iter(source))
```

This will use the iterator to build a list of `Blogs` in the rare cases that we actually want the entire sequence in memory. We can use the following to process each `Blog` individually, rendering it to create reST files:

```
with (Path.cwd()/"data"/"ch10_blog3.csv").open() as source:
    for b in blog_iter(source):
        with open(blog.title+'.rst','w') as rst_file:
            render(blog, rst_file)
```

We used the `blog_iter()` function to read each blog. After being read, it can be rendered into an `.rst` format file. A separate process can run `rst2html.py` to convert each blog into HTML.

We can easily add a filter to process only selected `Blog` instances. Rather than

simply rendering all the `Blog` instances, we can add an `if` statement to decide which `Blogs` should be rendered.

Let's see how to dump and load joined rows into a CSV file.

Dumping and loading joined rows into a CSV file

Joining the objects together means creating a collection where each row has a composite set of columns. The columns will be a union of the child class attributes and the parent class attributes. The file will have a row for each child. The parent attributes of each row will repeat the parent attribute values for the parent of that child. This involves a fair amount of redundancy, since the parent values are repeated with each individual child. When there are multiple levels of containers, this can lead to large amounts of repeated data.

The advantage of this repetition is that each row stands alone and doesn't belong to a context defined by the rows above it. We don't need a class discriminator. The parent values are repeated for each child object.

This works well for data that forms a simple hierarchy; each child has some parent attributes added to it. When the data involves more complex relationships, the simplistic parent-child pattern breaks down. In these examples, we've lumped the `Post` tags into a single column of text. If we tried to break the tags into separate columns, they would become children of each `Post`, meaning that the text of `Post` might be repeated for each tag. Clearly, this isn't a good idea!

The CSV column titles must be a union of all the available column titles. Because of the possibility of name clashes between the various row types, we'll qualify each column name with the class name. This will lead to column titles such as `'Blog.title'` and `'Post.title'`. This allows for the use of `DictReader` and `DictWriter` rather than the positional assignment of the columns. However, these qualified names don't trivially match the attribute names of the class definitions; this leads to somewhat more text processing to parse the column titles. Here's how we can write a joined row that contains parent as well as child attributes:

```
with (Path.cwd() / "data" / "ch10_blog5.csv").open("w", newline="") as target:
    wtr = csv.writer(target)
    wtr.writerow(
        ["Blog.title", "Post.date", "Post.title", "Post.tags", "Post.rst_text"]
    )
    for b in blogs:
```

```
        for p in b.entries:
            wtr.writerow([b.title, p.date, p.title, p.tags, p.rst_text])
```

We saw qualified column titles. In this format, each row now contains a union of the `Blog` attribute and the `Post` attributes. We can use the `b.title` and `p.title` attributes to include the blog title on each posting.

This data file layout is somewhat easier to prepare, as there's no need to fill unused columns with `None`. Since each column name is unique, we can easily switch to a `DictWriter` instead of a simple `csv.writer()`.

Rebuilding the blog entry becomes a two-step operation. The columns that represent the parent and `Blog` objects must be checked for uniqueness. The columns that represent the child and `Post` objects are built in the context of the most-recently-found parent. Here's a way to reconstruct the original container from the CSV rows:

```
def blog_iter2(source: TextIO) -> Iterator[Blog]:
    rdr = csv.DictReader(source)
    assert (
        set(rdr.fieldnames)
        == {"Blog.title", "Post.date", "Post.title", "Post.tags", "Post.rst_text"}
    )
    # Fetch first row, build the first Blog and Post.
    row = next(rdr)
    blog = blog_builder5(row)
    post = post_builder5(row)
    blog.append(post)

    # Fetch all subsequent rows.
    for row in rdr:
        if row["Blog.title"] != blog.title:
            yield blog
            blog = blog_builder5(row)
        post = post_builder5(row)
        blog.append(post)
    yield blog
```

The first row of data is used to build a `Blog` instance and the first `Post` in that `Blog`. The invariant condition for the loop that follows assumes that there's a proper `Blog` object. Having a valid `Blog` instance makes the processing logic much simpler. The `Post` instances are built with the following function:

```
import ast

def post_builder5(row: Dict[str, str]) -> Post:
    return Post(
        date=datetime.datetime.strptime(
            row["Post.date"],
            "%Y-%m-%d %H:%M:%S"),
        title=row["Post.title"],
```

```
        rst_text=row["Post.rst_text"],
        tags=ast.literal_eval(row["Post.tags"]),
    )
```

We mapped the individual columns in each row through a conversion to the parameters of the `constructor` class. This properly handles all of the type conversions from the CSV text into Python objects.

The `blog_builder5()` function is similar to the `post_builder5()` function. Since there are fewer attributes and no data conversion involved, it's not shown, and is left as an exercise for the reader.

Let's see how to dump and load using XML.

Dumping and loading with XML

Python's `xml` package includes numerous modules that parse XML files. There is also a **Document Object Model** (**DOM**) implementation that can produce an XML document. As with the previous `json` module, XML is not a complete persistence solution for Python objects. Because of the wide adoption of the XML files, however, it often becomes necessary to convert between Python objects and XML documents.

Working with XML files involves a manual mapping between Python objects and XML structures. We need to design the mapping carefully, remaining mindful of the constraints of XML's notation. This can be difficult because of the mismatch between the expressive powers of objects and the strictly hierarchical nature of an XML document.

The content of an XML attribute or tag is pure text. When loading an XML document, we'll need to convert these values into more useful types inside our applications. In some cases, the XML document might include attributes or tags to indicate the expected type.

If we are willing to put up with some limitations, we can use the `plistlib` module to emit some built-in Python structures as XML documents. We'll examine this module in `Chapter 14`, *Configuration Files and Persistence*, where we'll use it to load the configuration files.

The `json` module offers ways to extend the JSON encoding to include our customized classes; the `plistlib` module doesn't offer this additional hook.

When we look at dumping a Python object to create an XML document, there are three common ways to build the text:

- **Include XML output methods in our class design**: In this case, our classes emit strings that can be assembled into an XML document. This conflates serialization into the class in a potentially brittle design.
- **Use `xml.etree.ElementTree` to build the `ElementTree` nodes and return this structure**: This can be then be rendered as text. This is somewhat less

brittle because it builds an abstract document object model rather than text.

- **Use an external template and fill attributes into that template**: Unless we have a sophisticated template tool, this doesn't work out well. The `string.Template` class in the standard library is only suitable for very simple objects. More generally, Jinja2 or Mako should be used. This divorces XML from the class definitions.

There are some projects that include generic Python XML serializers. The problem with trying to create a generic serializer is that XML is extremely flexible; each application of XML seems to have unique **XML Schema Definition** (**XSD**) or **Document Type Definition** (**DTD**) requirements.

One open-XML document-design question is how to encode an atomic value. There are many choices. We could use a type-specific tag with an attribute name in the tag's attributes, such as `<int name="the_answer">42</int>`. Another possibility is to use an attribute-specific tag with the type in the tag's attributes: `<the_answer type="int">42</the_answer>`. We can also use nested tags: `<the_answer><int>42</int></the_answer>`. Or, we could rely on a separate schema definition to suggest that `the_answer` should be an integer and merely encode the value as text: `<the_answer>42</the_answer>`. We can also use adjacent tags: `<key>the_answer</key><int>42</int>`. This is not an exhaustive list; XML offers us a lot of choices.

When it comes to recovering Python objects from an XML document, there will be two steps. Generally, we have to parse the document to create the document object. Once this is available, we can then examine the XML document, assembling Python objects from the available tags.

Some web frameworks, such as Django, include XML serialization of Django-defined classes. This isn't a general serialization of arbitrary Python objects. The serialization is narrowly defined by Django's data-modeling components. Additionally, there are packages, such as `dexml`, `lxml`, and `pyxser`, as alternative bindings between Python objects and XML. Check out http://pythonhosted.org/dexml/api/dexml.html, http://lxml.de and http://coder.cl/products/pyxser/ for more information. A longer list of candidate packages can be found at: https://wiki.python.org/moin/PythonXml.

Now, let's see how to dump objects using string templates.

Dumping objects using string templates

One way to serialize a Python object into XML is by including a method to emit XML text. In the case of a complex object, the container must get the XML for each item inside the container. Here are two simple extensions to our microblog class structure that add the XML output capability as text:

```
from dataclasses import dataclass, field, asdict

@dataclass
class Blog_X:
    title: str
    entries: List[Post_X] = field(default_factory=list)
    underline: str = field(init=False)

    def __post_init__(self) -> None:
        self.underline = "="*len(self.title)

    def append(self, post: 'Post_X') -> None:
        self.entries.append(post)

    def by_tag(self) -> DefaultDict[str, List[Dict[str, Any]]]:
        tag_index: DefaultDict[str, List[Dict[str, Any]]] = defaultdict(list)
        for post in self.entries:
            for tag in post.tags:
                tag_index[tag].append(asdict(post))
        return tag_index

    def as_dict(self) -> Dict[str, Any]:
        return asdict(self)

    def xml(self) -> str:
        children = "\n".join(c.xml() for c in self.entries)
        return f"""\
<blog><title>{self.title}</title>
<entries>
{children}
<entries>
</blog>
"""
```

We've included several things in this dataclass-based definition. First, we defined the core attributes of a `Blog_X` object, a title and a list of entries. In order to make the entries optional, a field definition was provided to use the `list()` function as a factory for default values. In order to be compatible with the `Blog` class shown earlier, we've also provided an underline attribute that is built by the `__post_init__()` special method.

The `append()` method is provided at the `Blog_X` class level to be compatible with the `Blog` class from the xxx section. It delegates the work to the `entries` attribute. The `by_tag()` method can be used to build an index by hashtags.

The `as_dict()` method was defined for `Blog` objects, and emitted a dictionary built from the object. When working with dataclasses, the `dataclasses.asdict()` function does this for us. To be compatible with the `Blog` class, we wrapped the `asdict()` function into a method of the `Blog_X` dataclass.

The `xml()` method emits XML-based text for this object. It uses relatively unsophisticated f-string processing to inject values into strings. To assemble a complete entry, the `entries` collection is transformed into a series of lines, assigned to the `children` variable, and formatted into the resulting XML text.

The `Post_X` class definition is similar. It's shown here:

```
@dataclass
class Post_X:
    date: datetime.datetime
    title: str
    rst_text: str
    tags: List[str]
    underline: str = field(init=False)
    tag_text: str = field(init=False)

    def __post_init__(self) -> None:
        self.underline = "-"*len(self.title)
        self.tag_text = ' '.join(self.tags)

    def as_dict(self) -> Dict[str, Any]:
        return asdict(self)

    def xml(self) -> str:
        tags = "".join(f"<tag>{t}</tag>" for t in self.tags)
        return f"""\
<entry>
    <title>{self.title}</title>
    <date>{self.date}</date>
    <tags>{tags}</tags>
    <text>{self.rst_text}</text>
</entry>"""
```

This, too, has two fields that are created by the `__post_init__()` special method. It includes an `as_dict()` method to remain compatible with the `Post` class shown earlier. This method uses the `asdict()` function to do the real work of creating a dictionary from the dataclass object.

Both classes include highly class-specific XML output methods. These will emit

the relevant attributes wrapped in XML syntax. This approach doesn't generalize well. The Blog_X.xml() method emits a <blog> tag with a title and entries. The Post_X.xml() method emits a <post> tag with the various attributes. In both of these methods, subsidiary objects were created using "".join() or "\n".join() to build a longer string from shorter string elements. When we convert a Blog object into XML, the results look like this:

```
<blog><title>Travel</title>
<entries>
<entry>
    <title>Hard Aground</title>
    <date>2013-11-14 17:25:00</date>
    <tags><tag>#RedRanger</tag><tag>#Whitby42</tag><tag>#ICW</tag></tags>
    <text>Some embarrassing revelation. Including ☹ and ⚓</text>
</entry>
<entry>
    <title>Anchor Follies</title>
    <date>2013-11-18 15:30:00</date>
    <tags><tag>#RedRanger</tag><tag>#Whitby42</tag><tag>#Mistakes</tag></tags>
    <text>Some witty epigram.</text>
</entry>
<entries></blog>
```

This approach has two disadvantages:

- We've ignored the XML namespaces. That's a small change to the literal text for emitting the tags.
- Each class would also need to properly escape the <, &, >, and " characters into the <, >, &, and " XML entities. The html module includes the html.escape() function which does this.

This emits proper XML; it can be relied upon to work; but it isn't very elegant and doesn't generalize well.

In the next section, we'll see how to dump objects with xml.etree.ElementTree.

Dumping objects with xml.etree.ElementTree

We can use the `xml.etree.ElementTree` module to build `Element` structures that can be emitted as XML. It's challenging to use `xml.dom` and `xml.minidom` for this. The DOM API requires a top-level document that then builds individual elements. The presence of this necessary context object creates clutter when trying to serialize a simple class with several attributes. We have to create the document first and then serialize all the elements of the document, providing the document context as an argument.

Generally, we'd like each class in our design to build a top-level element and return that. Most top-level elements will have a sequence of subelements. We can assign text as well as attributes to each element that we build. We can also assign a *tail*, which is the extraneous text that follows a closed tag. In some content models, this is just whitespace. Because of the long name, it might be helpful to import `ElementTree` in the following manner:

```
import xml.etree.ElementTree as XML
```

Here are two extensions to our microblog class structure that add the XML output capability as the `Element` instances. We can use the following extension to the `Blog_X` class:

```
import xml.etree.ElementTree as XML
from typing import cast

class Blog_E(Blog_X):

    def xmlelt(self) -> XML.Element:
        blog = XML.Element("blog")
        title = XML.SubElement(blog, "title")
        title.text = self.title
        title.tail = "\n"
        entities = XML.SubElement(blog, "entries")
        entities.extend(cast('Post_E', c).xmlelt() for c in self.entries)
        blog.tail = "\n"
        return blog
```

We can use the following extension to the `Post_X` class:

```
class Post_E(Post_X):

    def xmlelt(self) -> XML.Element:
        post = XML.Element("entry")
        title = XML.SubElement(post, "title")
        title.text = self.title
        date = XML.SubElement(post, "date")
        date.text = str(self.date)
        tags = XML.SubElement(post, "tags")
        for t in self.tags:
            tag = XML.SubElement(tags, "tag")
            tag.text = t
        text = XML.SubElement(post, "rst_text")
        text.text = self.rst_text
        post.tail = "\n"
        return post
```

We've written highly class-specific XML output methods. These will build the Element objects that have the proper text values.

There's no fluent shortcut for building the subelements. We have to insert each text item individually.

In the Blog.xmlelt() method, we were able to perform Element.extend() to put all of the individual post entries inside the <entry> element. This allows us to build the XML structure flexibly and simply. This approach can deal gracefully with the XML namespaces. We can use the QName class to build qualified names for XML namespaces. The ElementTree module correctly applies the namespace qualifiers to the XML tags. This approach also properly escapes the <, &, >, and " characters into the <, >, &, and " XML entities. The XML output from these methods will mostly match the previous section. The whitespace will be different.

To build the final output, we use two additional features of the element tree module. It will look like this snippet:

```
tree = XML.ElementTree(travel5.xmlelt())
text = XML.tostring(tree.getroot())
```

The travel5 object is an instance of Blog_E. The result of evaluating travel5.xmlelt() is an XML.Element; this is wrapped into a complete XML.ElementTree object. The tree's root object can be transformed into a valid XML string and printed or saved to a file.

Let's see how to load XML documents.

Loading XML documents

Loading Python objects from an XML document is a two-step process. First, we need to parse the XML text to create the document objects. Then, we need to examine the document objects to produce Python objects. As noted previously, the tremendous flexibility of XML notation means that there isn't a single XML-to-Python serialization.

One approach to walk through an XML document involves making XPath-like queries to locate the various elements that were parsed. Here's a function to walk through an XML document, emitting the `Blog` and `Post` objects from the available XML:

```
def build_blog(document: XML.ElementTree) -> Blog_X:
    xml_blog = document.getroot()
    blog = Blog_X(xml_blog.findtext("title"))
    for xml_post in xml_blog.findall("entries/entry"):
        optional_tag_iter = (
            t.text for t in xml_post.findall("tags/tag")
        )
        tags = list(
            filter(None, optional_tag_iter)
        )
        post = Post_X(
            date=datetime.datetime.strptime(
                xml_post.findtext("date"), "%Y-%m-%d %H:%M:%S"
            ),
            title=xml_post.findtext("title"),
            tags=tags,
            rst_text=xml_post.findtext("rst_text"),
        )
        blog.append(post)
    return blog
```

This function traverses a `<blog>` XML document. It locates the `<title>` tag and gathers all of the text within that element to create the top-level `Blog` instance. It then locates all the `<entry>` subelements found within the `<entries>` element. These are used to build each `Post` object. The various attributes of the `Post` object are converted individually. The text of each individual `<tag>` element within the `<tags>` element is turned into a list of text values. The date is parsed from its text representation. The `Post` objects are each appended to the overall `Blog` object. This *manual* mapping from XML text to Python objects is an essential feature of parsing XML documents.

The value of the `(t.text for t in xml_post.findall("tags/tag"))` generator expression does not have the `Iterator[str]` type. It turns out that the values of the `t.text` attribute have an `Optional[str]` type. The resulting expression would create a list with a type hint of `List[Optional[str]]`, which isn't directly compatible with the `Post_X` class.

There are two resolutions to this problem: we could expand the definition in `Post_X` to use `List[Optional[str]]`. This could lead to a need to filter out `None` objects elsewhere in the application. Instead, we pushed the removal of `None` objects into this parser. The `filter(None, iterable)` function will remove all None values from the iterable; this transforms a value with the `List[Optional[str]]` type hint into a value with the `List[str]` type hint.

This kind of transformation and filtering is an essential part of XML processing. Each distinct data type or structure will have to be serialized as an XML-compatible string and deserialized from the XML string. XML provides a structure, but the details of serialization remain an essential part of the Python application programming.

Summary

We looked at a number of ways to serialize Python objects. We can encode our class definitions in notations, including JSON, YAML, pickle, XML, and CSV. Each of these notations has a variety of advantages and disadvantages.

These various library modules generally work around the idea of loading objects from an external file or dumping objects into a file. These modules aren't completely consistent with each other, but they're very similar, allowing us to apply some common design patterns.

Using CSV and XML tends to expose the most difficult design problems. Our class definitions in Python can include object references that don't have a good representation in the CSV or XML notation.

Design considerations and tradeoffs

There are many ways to serialize and persist Python objects. We haven't seen all of them yet. The formats in this section are focused on two essential use cases:

- **Data interchange with other applications**: We might be publishing data for other applications or accepting data from other applications. In this case, we're often constrained by the other applications' interfaces. Often, JSON and XML are used by other applications and frameworks as their preferred form of data interchange. In some cases, we'll use CSV to exchange data.
- **Persistent data for our own applications**: In this case, we're usually going to choose `pickle` because it's complete and is already part of the Python Standard Library. However, one of the important advantages of YAML is its readability; we can view, edit, and even modify the file.

When working with each of these formats, we have a number of design considerations. First and foremost, these formats are biased towards serializing a single Python object. It might be a list of other objects, but it is essentially a single object. JSON and XML, for example, have ending delimiters that are written after the serialized object. For persisting individual objects from a larger domain, we can look at `shelve` and `sqlite3` in `Chapter 11`, *Storing and Retrieving Objects via Shelve*, and `Chapter 12`, *Storing and Retrieving Objects via SQLite*.

JSON is a widely-used standard, but it's inconvenient for representing complex Python classes. When using JSON, we need to be cognizant of how our objects can be reduced to a JSON-compatible representation. JSON documents are human-readable. JSON's limitations make it potentially secure for the transmission of objects through the internet.

YAML is not as widely used as JSON, but it solves numerous problems in serialization and persistence. YAML documents are human-readable; for editable configuration files, YAML is ideal. We can make YAML secure using the safe-load options.

Pickle is ideal for the simple, fast, local persistence of Python objects. It is a compact notation for the transmission from Python to Python. CSV is a widely-

used standard. Working out representations for Python objects in CSV notation is challenging. When sharing data in CSV notation, we often end up using `NamedTuple` objects in our applications. We have to design a mapping from Python to CSV and CSV to Python.

XML is another widely-used notation for serializing data. XML is extremely flexible, leading to a wide variety of ways to encode Python objects in XML notation. Because of the XML use cases, we often have external specifications in the form of an XSD or DTD. The process for parsing XML to create Python objects is always rather complex.

Because each CSV row is largely independent of the others, CSV allows us to encode or decode extremely large collections of objects. For this reason, CSV is often handy for encoding and decoding gargantuan collections that can't fit into memory.

In some cases, we have a hybrid design problem. When reading most modern spreadsheet files, we have the CSV row-and-column problem wrapped in the XML parsing problem. Consider, for example, OpenOffice. ODS files are zipped archives. One of the files in the archive is the `content.xml` file. Using an XPath search for `body/spreadsheet/table` elements will locate the individual tabs of the spreadsheet document. Within each table, we'll find the `table-row` elements that (usually) map to Python objects. Within each row, we'll find the `table-cell` elements that contain the individual values that build up the attributes of an object.

Schema evolution

When working with persistent objects, we have to address the problem of schema evolution. Our objects have a dynamic state and a static class definition. We can easily persist the dynamic state. Our class definitions are the schema for the persistent data. The class, however, is not *absolutely* static. When a class changes, we need to make a provision to load data that was dumped by the previous release of our application.

It's best to think of external file compatibility to distinguish between major and minor release numbers. A major release should mean that a file is no longer compatible and a conversion must be done. A minor release should mean that the file formats are compatible and no data conversion will be involved in the upgrade.

One common approach is to include the major version number in the file extension. We might have filenames that end in `.json2` or `.json3` to indicate which format of data is involved. Supporting multiple versions of a persistent file format often becomes rather complex. To provide a seamless upgrade path, an application should be able to decode previous file formats. Often, it's best to persist data in the latest and greatest file format, even if the other formats are supported for input.

In the next chapters, we'll address serialization that's not focused on a single object. The `shelve` and `sqlite3` modules give us ways to serialize a multitude of distinct objects. After that, we'll return to using these techniques for **REpresentational State Transfer** (**REST**) to transmit objects from process to process. Also, we'll use these techniques yet again to process the configuration files.

Looking forward

In `Chapter 11`, *Storing and Retrieving Objects via Shelve*, and `Chapter 12`, *Storing and Retrieving Objects via SQLite*, we'll look at two common approaches to making larger collections of persistent objects. These two chapters show us different approaches we can use to create a database of Python objects.

In `Chapter 13`, *Transmitting and Sharing Objects*, we'll apply these serialization techniques to the problem of making an object available in another process. We'll focus on RESTful web services as a simple and popular way to transmit an object among processes.

In `Chapter 14`, *Configuration Files and Persistence*, we'll apply these serialization techniques yet again. In this case, we'll use representations such as JSON and YAML to encode the configuration information for an application.

Storing and Retrieving Objects via Shelve

There are many applications where we need to persist objects individually. The techniques we looked at in `Chapter 10`, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML*, were biased toward handling a single object. Sometimes, we need to persist separate, individual objects from a larger domain.

Applications with persistent objects may demonstrate four use cases, summarized as the **CRUD operations**: create, retrieve, update, and delete. The idea here is any of these operations may be applied to any object in the domain; this leads to the need for a more sophisticated persistence mechanism than a monolithic load or dump of all the objects into a file. In addition to squandering memory, simple loads and dumps are often less efficient than fine-grained, distinct, object-by-object storage.

Using more sophisticated storage will lead us to look more closely at the allocation of responsibility. Separating the various concerns give us overall design patterns for the architecture of the application software. One example of these higher-level design patterns is the **multi-tier architecture**:

- **Presentation layer**: This includes web browsers or mobile apps. It can also include a **graphical user interface** (**GUI**) for a locally-installed application. In some cases, this can also be a textual command-line interface.
- **Application layer**: While this is often based on a web server, it may also be a portion of a locally-installed software. The application tier can be usefully subdivided into a processing layer and a data model layer. The processing layer involves the classes and functions that embody an application's behavior. The data model layer defines the problem domain's object model.
- **Data layer**: This can be further subdivided into an access layer and a persistence layer. The access layer provides uniform access to persistent objects. The persistence layer serializes objects and writes them to the persistent storage. This is where the more sophisticated storage techniques are implemented.

Because some of these tiers can be subdivided, there are a number of variations on this model. It can be called a three-tier architecture to recognize the clearest distinctions. It can also be called an ***n*-tier architecture** to permit fine degrees of hair-splitting.

The data layer can use modules, such as the `shelve` module, for persistence. This module defines a mapping-like container in which we can store objects. Each stored object is pickled and written to a file. We can also unpickle and retrieve any object from the file. The `shelve` module relies on the `dbm` module to save and retrieve objects.

This section will focus on the access and persistence layers within the overall data layer. The interface between these layers will be a class interface within a single application. We'll focus on the simple class-to-class interface in this chapter. We'll look at a network-based interface to a data layer in `Chapter 13`, *Transmitting and Sharing Objects*, using REST.

In this chapter, we will cover the following topics:

- Analyzing persistent object use cases
- Creating a shelf
- Designing shelvable objects
- Searching, scanning, and querying
- Designing an access layer for `shelve`
- Creating indexes to improve efficiency
- Adding yet more index maintenance
- The writeback alternative to index updates

Technical requirements

The code files for this chapter are available at `https://git.io/fj2Ur`.

Analyzing persistent object use cases

The persistence mechanisms we looked at in `Chapter 10`, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML*, focused on reading and writing a compact file with the serialized representation of one or more objects. If we wanted to update any part of the file, we were forced to replace the entire file. This is a consequence of using a compact notation for the data: it's difficult to reach the position of an individual object within a file, and it's difficult to replace an object if the size changes. Rather than addressing these difficulties with smart, complex algorithms, all of the data was serialized and written.

When we have a larger problem domain with many persistent, independent, and mutable objects, we introduce some additional depth to the use cases:

- We may not want to load all of the objects into the memory at one time. For many *big data* applications, it might be impossible to load all the objects into the memory at one time.
- We may be updating only small subsets – or individual instances – from our domain of objects. Loading and then dumping all the objects to update one object is relatively inefficient processing.
- We may not be dumping all the objects at one time; we may be accumulating objects incrementally. Some formats, such as YAML and CSV, allow us to append themselves to a file with little complexity. Other formats, such as JSON and XML, have terminators that make it difficult to simply append to a file.

There are still more features we might consider. It's common to conflate serialization, persistence, transactional consistency, as well as concurrent write access into a single umbrella concept of *database*. The `shelve` module is not a comprehensive database solution by itself. The underlying `dbm` module used by `shelve` does not directly handle concurrent writes. It doesn't handle multi-operation transactions either. It's possible to use low-level OS locking on the files to tolerate concurrent updating. For concurrent write access, it's better to either use a proper database or a RESTful data server. See `Chapter 12`, *Storing and Retrieving Objects via SQLite*, and `Chapter 13`, *Transmitting and Sharing Objects*, for more information.

Let's take a look at the ACID properties in the next section.

The ACID properties

Our design must consider how the **ACID properties** apply to our `shelve` database. Often, an application will make changes in bundles of related operations; the bundle should change the database from one consistent state to the next consistent state. The intent of a transactional bundle is to conceal any intermediate states that may not be consistent with other users of the data.

An example of multiple-operation transactions could involve updating two objects so that a total is kept invariant. We might be deducting funds from one financial account and depositing into another. The overall balance must be held constant for the database to be in a consistent, valid state.

The ACID properties characterize how we want the database transactions to behave as a whole. There are four rules that define our expectations:

- **Atomicity**: A transaction must be atomic. If there are multiple operations within a transaction, either all the operations should be completed or none of them should be completed. It should never be possible to view a partially completed transaction.
- **Consistency**: A transaction must assure consistency of the overall database. It must change the database from one valid state to another. A transaction should not corrupt the database or create inconsistent views among concurrent users. All users see the same net effect of completed transactions.
- **Isolation**: Each transaction should operate as if it processed in complete isolation from all other transactions. We can't have two concurrent users interfering with each other's attempted updates. We should always be able to transform concurrent access into (possibly slower) serial access and the database updates will produce the same results. Locks are often used to achieve this.
- **Durability**: The changes to the database should persist properly in the filesystem.

When we work with in-memory Python objects, clearly, we can get **ACI,** but we don't get **D**. In-memory objects are not durable by definition. If we attempt to

use the `shelve` module from several concurrent processes without locking or versioning, we may get only **D**, but lose the **ACI** properties.

The `shelve` module doesn't provide direct support for atomicity; it lacks an integrated technique to handle transactions that consist of multiple operations. If we have multiple-operation transactions and we need atomicity, we must ensure that they all work or all fail as a unit. This can involve saving the state prior to a `try:` statement; in the event of a problem, the exception handlers must restore the previous state of the database.

The `shelve` module doesn't guarantee durability for all kinds of changes. If we place a mutable object onto the shelf and then change the object in memory, the persistent version on the shelf file will not change *automatically*. If we're going to mutate shelved objects, our application must be explicit about updating the shelf object. We can ask a shelf object to track changes via the *writeback mode*, but using this feature can lead to poor performance.

These missing features are relatively simple to implement with some additional locking and logging. They aren't default features of a `shelf` instance. When full ACID capabilities are needed, we often switch to other forms of persistence. When an application doesn't need the full ACID feature set, however, the `shelve` module can be very helpful.

In the next section, we'll see how to create a shelf.

Creating a shelf

The first part of creating a shelf is done using a module-level function, `shelve.open()`, to create a persistent shelf structure. The second part is closing the file properly so that all changes are written to the underlying filesystem. We'll look at this in a more complete example, in the *Designing an access layer for shelve* section.

Under the hood, the `shelve` module is using the `dbm` module to do the real work of opening a file and mapping from a key to a value. The `dbm` module itself is a wrapper around an underlying DBM-compatible library. Consequently, there are a number of potential implementations for the `shelve` features. The good news is that the differences among the `dbm` implementations are largely irrelevant.

The `shelve.open()` module function requires two parameters: the filename and the file access mode. Often, we want the default mode of `'c'` to open an existing shelf or create one if it doesn't exist.

The alternatives are for specialized situations:

- `'r'` is a read-only shelf.
- `'w'` is a read-write shelf that *must* exist or an exception will be raised.
- `'n'` is a new, empty shelf; any previous version will be overwritten.

It's absolutely essential to close a shelf to be sure that it is properly persisted to disk. The shelf is not a context manager itself, but the `contextlib.closing()` function should always be used to make sure that the shelf is closed. For more information on context managers, see Chapter 6, *Using Callables and Contexts.*

Under some circumstances, we might also want to explicitly synchronize a shelf to a disk without closing the file. The `shelve.sync()` method will persist changes prior to a close. An idealized life cycle looks something like the following code:

```
import shelve
from contextlib import closing
from pathlib import Path

db_path = Path.cwd() / "data" / "ch11_blog"
with closing(shelve.open(str(db_path))) as shelf:
```

```
    process(shelf)
```

We've opened a shelf and provided that open shelf to some function that does the real work of our application. When this process is finished, the context will ensure that the shelf is closed. If the `process()` function raises an exception, the shelf will still be properly closed.

Let's see how to design shelvable objects.

Designing shelvable objects

If our objects are relatively simple, putting them on a shelf will be trivial. For objects that are not complex containers or large collections, we only have to work out a key-to-value mapping. For objects that are more complex – typically objects that contain other objects – we have to make some additional design decisions regarding the granularity of access and references among objects. We'll look at the simple case first, where all we have to design is the key that is used to access our objects. Then, we'll look at the more complex cases, where granularity and object references come into play.

Let's see how to design objects with type hints.

Designing objects with type hints

The Python type hints provide considerable help in defining objects for a shelf. Throughout this chapter, we'll emphasize the use of the `@dataclass` decorator to create objects suitable for persistence.

The data class concept is helpful because it makes the attributes that define object state extremely clear. The attributes are not concealed inside method definitions. Without `@dataclass`, the attributes are often implied by the `__init__()` method. In some classes, however, attributes are defined dynamically, making the attributes inconsistent and leading to possible problems in recovering object state from a shelved representation.

The `pickle` module is used to do the serialization of the objects. For more information on object pickling, see `Chapter 10`, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML*.

In the following sections, we'll look at ways to define the keys used to uniquely identify objects in a shelf collection. A single unique key is essential. In some problem domains, there will be an attribute, or combination of attributes, that is unique. In spite of this, it's often advantageous to create a surrogate key that is generated by the application and assigned to each persistent object to assure uniqueness.

Let's see how to design keys for our objects.

Designing keys for our objects

The `shelve` and `dbm` modules provide immediate access to any object in an arbitrarily huge universe of objects. The `shelve` module creates a mapping that is much like a dictionary. The shelf mapping exists on persistent storage, so any object we put onto the shelf will be serialized and saved.

We must identify each shelved object with a unique key. String values are a common choice for keys. This imposes some design considerations on our classes to provide an appropriately unique key. In some cases, the problem domain will have an attribute that is an obvious unique key. In that case, we can simply use that attribute to construct this key. For example, if our class has a unique attribute value, `key_attribute`, we can use `shelf[object.key_attribute] = object`. This is the simplest case and sets the pattern for more complex cases.

When our application problem doesn't offer an appropriate unique key, we'll have to fabricate a surrogate key value. This problem arises frequently when every attribute of an object is potentially mutable or potentially non-unique. In this case, we may have to create a surrogate key because there is no value, nor any combination of values, that is unique.

Our application may have non-string values that are candidates for primary keys. For example, we might have a `datetime` object, or a number. In these cases, we might want to encode the value as a string.

In cases where there is no obvious primary key, we can try to locate a combination of values that create a unique **composite key**. This can become complex, because now the key is not atomic, and a change to any part of the key may create data update problems.

It's often simplest to follow a design pattern called a **surrogate key**. This key doesn't depend on data within an object; it's a surrogate for the object. This means that any of the attributes of the object can be changed without leading to complications or restrictions. Python's internal object IDs are an example of a kind of surrogate key.

The string representation of a shelf key can follow this pattern: `class_name:oid`. The key string includes the class of the object, `class_name`, paired with the unique identifier for an instance of the class, `oid`. We can easily store diverse classes of objects in a single shelf using keys of this form. Even when we think there will be only one type of object in the shelf, this format is still helpful to save a namespace for indexes, administrative metadata, and future expansion.

When we have a suitable natural key, we might do something like this to persist objects in the shelf:

```
shelf[f"{object.__class__.__name__}:{object.key_attribute}"] = object
```

This uses an f-string to create a key with a distinct class name along with the unique key value. This string identifier must be unique for each object. For surrogate keys, we'll need to define some kind of generator to emit distinct values.

The next section discusses how to generate surrogate keys for objects.

Generating surrogate keys for objects

One way to generate unique surrogate keys is to use an integer counter. To be sure that we keep this counter properly updated, we will store it in the shelf along with the rest of our data. Even though Python has an internal object ID, we should not use Python's internal identifier for a surrogate key. Python's internal ID numbers have no guarantees of any kind.

As we're going to add some administrative objects to our shelf, we must give these objects unique keys with a distinctive prefix. We'll use `_DB`. This will be a class name for the administrative objects in our shelf. The design decisions for these administrative objects are similar to the design of the application objects. We need to choose the granularity of storage. We have two choices:

- **Coarse-grained**: We can create a single `dict` object with all of the administrative overheads for surrogate key generations. A single key, such as `_DB:max`, can identify this object. Within this `dict`, we could map class names to the maximum identifier values used. Every time we create a new object, we assign the ID from this mapping and then also replace the mapping in the shelf. We'll show the coarse-grained solution in the following *Designing a class with a simple key* section.
- **Fine-grained**: We can add many items to the database, each of which has the maximum key value for a different class of objects. Each of these additional key items has the form of `_DB:max:class`. The value for each of these keys is just an integer, the largest sequential identifier assigned so far for a given class.

An important consideration here is that we've separated the key design from the class design for our application's classes. We can (and should) design our application objects as simply as possible. We should add just enough overhead to make `shelve` work properly, but no more.

Let's see how to design a class with a simple key.

Designing a class with a simple key

It is helpful to store the `shelve` key as an attribute of a shelved object. Keeping the key in the object makes the object easier to delete or replace. Clearly, when creating an object, we'll start with a keyless version of the object until it's stored in the shelf. Once stored, the Python object needs to have a key attribute set so that each object in the memory contains a correct key.

When retrieving objects, there are two use cases. We might want a specific object that is known by the key. In this case, the shelf will map the key to the object. We might also want a collection of related objects not known by their keys, but perhaps known by the values of some other attributes. In this case, we'll discover the keys of objects through some kind of search or query. We'll look at the search algorithms in the following *Designing CRUD operations for complex objects* section.

To support saving the shelf keys in objects, we'll add an `_id` attribute to each object. It will keep the `shelve` key in each object that has been put onto the shelf or retrieved from the shelf. This will simplify managing objects that need to be replaced in or removed from the shelf. A surrogate key won't have any method functions, for example, and it is never part of the processing layer of the application tier or the presentation tier. Here's a definition for an overall `Blog`:

```
from dataclasses import dataclass, asdict, field
@dataclass
class Blog:

    title: str
    entries: List[Post] = field(default_factory=list)
    underline: str = field(init=False, compare=False)

    # Part of the persistence, not essential to the class.
    _id: str = field(default="", init=False, compare=False)

    def __post_init__(self) -> None:
        self.underline = "=" * len(self.title)
```

We've provided the essential `title` attribute. The `entries` attribute is optional, and has a default value of an empty list. `underline` is computed as a string with a length that matches the title; this makes some part of reStructuredText formatting slightly simpler.

We can create a `Blog` object in the following manner:

```
>>> b1 = Blog(title="Travel Blog")
```

This will have an empty list of individual postings within the blog. When we store this simple object in the shelf, we can do operations including the following:

```
>>> import shelve
>>> from pathlib import Path
>>> shelf = shelve.open(str(Path.cwd() / "data" / "ch11_blog"))
>>> b1._id = 'Blog:1'
>>> shelf[b1._id] = b1
```

We started by opening a new shelf. The file will end up being called `ch11_blog.db`. We put a key, `Blog:1`, into our `Blog` instance, `b1`. We stored that `Blog` instance in the shelf using the key given in the `_id` attribute.

We can fetch the item back from the shelf like this:

```
>>> shelf['Blog:1']
Blog(title='Travel Blog', entries=[], underline='===========', _id='Blog:1')
>>> shelf['Blog:1'].title
'Travel Blog'
>>> shelf['Blog:1']._id
'Blog:1'
>>> list(shelf.keys())
['Blog:1']
>>> shelf.close()
```

When we refer to `shelf['Blog:1']`, it will fetch our original `Blog` instance from the shelf. We've put only one object on the shelf, as we can see from the list of keys. Because we closed the shelf, the object is persistent. We can quit Python, start it back up again, open the shelf, and see that the object remains on the shelf, using the assigned key. Previously, we mentioned a second use case for retrieval—locating an item without knowing the key. Here's a search that locates all the blogs with a given title:

```
>>> path = Path.cwd() / "data" / "ch11_blog"
>>> shelf = shelve.open(str(path))
>>> results = (shelf[k]
...     for k in shelf.keys()
...     if k.startswith('Blog:') and shelf[k].title == 'Travel Blog'
... )
>>> list(results)
[Blog(title='Travel Blog', entries=[], underline='===========', _id='Blog:1')]
```

We opened the shelf to get access to the objects. The `results` generator expression

examines each item in the shelf to locate those items where the key starts with `'Blog:'`, and the object's title attribute is the `'Travel Blog'` string.

What's important is that the key, `'Blog:1'`, is stored within the object itself. The `_id` attribute ensures that we have the proper key for any item that our application is working with. We can now mutate any attribute of the object (except the key) and replace it in the shelf using its original key.

Now, let's see how to design classes for containers or collections.

Designing classes for containers or collections

When we have more complex containers or collections, we have more complex design decisions to make. One question is about the **granularity** of our shelved objects.

When we have an object, such as `Blog`, which is a container, we can persist the entire container as a single, complex object on our shelf. To an extent, this might defeat the purpose of having multiple objects on a shelf in the first place. Storing large containers involves coarse-grained storage. If we change a contained object, the entire container must be serialized and stored. If we wind up effectively pickling the entire universe of objects in a single container, why use `shelve`? We must strike a balance that is appropriate to the application's requirements.

One alternative is to decompose the collection into separate, individual items. In this case, our top-level `Blog` object won't be a proper Python container anymore. The parent might refer to each child with a collection of keys. Each child object could refer to the parent by the key. This use of keys is unusual in object-oriented design. Normally, objects simply contain references to other objects. When using `shelve` (or other databases), we can be forced to use indirect references by the key.

Each child will now have two keys: its own *primary key*, plus a *foreign key* that is the primary key of the parent object. This leads to a second design question about representing the key strings for the parents and their children.

The next section shows how to refer to objects via foreign keys.

Referring to objects via foreign keys

The key that we use to uniquely identify an object is its *primary key*. When child objects refer to a parent object, we have additional design decisions to make. How do we structure the children's primary keys? There are two common design strategies for child keys, based on the kind of dependence that exists between the classes of objects:

- `"Child:`*`cid`*`"`: We can use this when we have children that can exist independently of an owning parent. For example, an item on an invoice refers to a product; the product can exist even if there's no invoice item for the product.
- `"Parent:`*`pid`*`:Child:`*`cid`*`"`: We can use this when the child cannot exist without a parent. A customer address doesn't exist without a customer to contain the address in the first place. When the children are entirely dependent on the parent, the child's key can contain the owning parent's ID to reflect this dependency.

As with the parent class design, it's easiest if we keep the primary key and all foreign keys associated with each child object. We suggest not initializing them in the `__init__()` method, as they're just features of persistence. Here's the general definition for `Post` within `Blog`:

```
import datetime
from dataclasses import dataclass, field, asdict
from typing import List
@dataclass
class Post:
    date: datetime.datetime
    title: str
    rst_text: str
    tags: List[str]
    underline: str = field(init=False)
    tag_text: str = field(init=False)

    # Part of the persistence, not essential to the class.
    _id: str = field(default='', init=False, repr=False, compare=False)
    _blog_id: str = field(default='', init=False, repr=False, compare=False)

    def __post_init__(self) -> None:
        self.underline = "-" * len(self.title)
        self.tag_text = " ".join(self.tags)
```

We've provided several attributes for each microblog post. The `asdict()` function

of the `dataclasses` module can be used with a template to provide a dictionary usable to create JSON notation. We've avoided mentioning the primary key or any foreign keys for `Post`. Here are two examples of the `Post` instances:

```
p2 = Post(date=datetime.datetime(2013,11,14,17,25),
        title="Hard Aground",
        rst_text="""Some embarrassing revelation. Including ☹ and ⚓""",
        tags=("#RedRanger", "#Whitby42", "#ICW"),
        )

p3 = Post(date=datetime.datetime(2013,11,18,15,30),
        title="Anchor Follies",
        rst_text="""Some witty epigram. Including < & > characters.""",
        tags=("#RedRanger", "#Whitby42", "#Mistakes"),
        )
```

We can now associate these two post objects with their owning blog object by setting attributes. We'll do this through the following steps:

1. Open the shelf and retrieve the parent `Blog` object. Save it in the `owner` variable so we have access to the `_id` attribute:

```
>>> import shelve
>>> shelf = shelve.open("blog")
>>> owner = shelf['Blog:1']
```

2. Assign this owner's key to each `Post` object and persist the objects. Put the parent information into each `Post`. We used the parent information to build the primary key. For this dependent kind of key, the `_parent` attribute value is redundant; it can be deduced from the key. If we used an independent key design for `Posts`, however, `_parent` would not be duplicated in the key:

```
>>> p2._blog_id = owner._id
>>> p2._id = p2._blog_id + ':Post:2'
>>> shelf[p2._id]= p2

>>> p3._blog_id = owner._id
>>> p3._id = p3._blog_id + ':Post:3'
>>> shelf[p3._id]= p3
```

When we look at the keys, we can see the `Blog` plus both `Post` instances:

```
>>> list(shelf.keys())
['Blog:1:Post:3', 'Blog:1', 'Blog:1:Post:2']
```

When we fetch any child `Post`, we'll know the proper parent `Blog` for the individual posting:

```
>>> p2._parent 'Blog:1'
```

```
>>> p2._id 'Blog:1:Post:2'
```

Following the keys the other way – from parent, `Blog`, down to its child, `Post` – becomes a matter of locating the matching keys in the shelf collection.

How to design CRUD operations for complex objects is discussed in the next section.

Designing CRUD operations for complex objects

When we decompose a larger collection into a number of separate fine-grained objects, we will have multiple classes of objects on the shelf. Because they are independent objects, they will lead to separate sets of CRUD operations for each class. In some cases, the objects are independent, and operations on an object of one class have no impact outside that individual object. In some relational database products, these become *cascading* operations. Removing a `Blog` entry can cascade into removing the related `Post` entries.

In our previous example, the `Blog` and `Post` objects have a dependency relationship. The `Post` objects are children of a parent `Blog`; the child can't exist without the parent. When we have these dependent relationships, we have a more entangled collection of operations to design. Here are some of the considerations:

- Consider the following for CRUD operations on independent (or parent) objects:
    - We may create a new, empty parent, assigning a new primary key to this object. We can later assign children to this parent. Code such as `shelf['parent:'+object._id] = object` creates a parent object in the shelf.
    - We may update or retrieve this parent without any effect on the children. We can perform `shelf['parent:'+some_id]` on the right side of the assignment to retrieve a parent. Once we have the object, we can perform `shelf['parent:'+object._id] = object` to persist a change.
    - Deleting the parent can lead to one of two behaviors. One choice is to cascade the deletion to include all the children that refer to the parent. Alternatively, we may write code to prohibit the deletion of parents that still have child references. Both are sensible, and the choice is driven by the requirements imposed by the problem domain.
- Consider the following for CRUD operations on dependent (or child) objects:
    - We can create a new child that refers to an existing parent. We must also decide what kind of keys we want to use for children and parents.

- We can update, retrieve, or delete the child outside the parent. This can include assigning the child to a different parent.

As the code to replace an object is the same as the code to update an object, half of the CRUD processing is handled through the simple assignment statement. Deletion is done with the `del` statement. The issue of deleting children associated with a parent might involve a retrieval to locate the children. What's left, then, is an examination of retrieval processing, which can be a bit more complex.

Searching, scanning, and querying are discussed in the next section.

Searching, scanning, and querying

A search can be inefficient if we examine all of the objects in a database, and apply a filter. We'd prefer to work with a more focused subset of items. We'll look at how we can create more useful indices in the *Creating indexes to improve efficiency* section. The fallback plan of brute-force scanning all the objects, however, always works. For searches that occur rarely, the computation required to create a more efficient index may not be worth the time saved.

Don't panic; searching, scanning, and querying are synonyms. We'll use the terms interchangeably.

When a child class has an independent-style key, we can scan a shelf for all instances of some `Child` class using an iterator over the keys. Here's a generator expression that locates all the children:

```
children = (shelf[k]
    for k in shelf.keys()
    if k.startswith("Child:"))
```

This looks at every single key in the shelf to pick the subset that begins with `"Child:"`. We can build on this to apply more criteria by using a more complex generator expression:

```
children_by_title = (c
    for c in children
    if c.startswith("Child:") and c.title == "some title")
```

We've used a nested generator expression to expand on the initial `children` query, adding criteria. Nested generator expressions such as this are remarkably efficient in Python. This does not make two scans of the database. It's a single scan with two conditions. Each result from the inner generator feeds the outer generator to build the result.

When a child class has a dependent-style key, we can search the shelf for children of a specific parent using an iterator with a more complex matching rule. Here's a generator expression that locates all the children of a given parent:

```
children_of = (shelf[k]
    for k in shelf.keys()
```

```
    if k.startswith(parent+":Child:"))
```

This dependent-style key structure makes it particularly easy to remove a parent and all children in a simple loop:

```
query = (key
    for key in shelf.keys()
    if key.startswith(parent))
for k in query:
    del shelf[k]
```

When using hierarchical `"Parent:pid:Child:cid"` keys, we do have to be careful when separating parents from their children. With this multi-part key, we'll see lots of object keys that start with `"Parent:pid"`. One of these keys will be the proper parent, simply `"Parent:pid"`. The other keys will be children with `"Parent:pid:Child:cid"`. We have three kinds of conditions that we'll often use for these brute-force searches:

- `key.startswith(f"Parent:{pid}")`: Finds a union of parents and children; this isn't a common requirement.
- `key.startswith(f"Parent:{pid}:Child:")`: Finds children of the given parent. An alternative to `startswith()` is a regular expression, such as `r"^(Parent:\d+):(Child:\d+)$"`, to match the keys.
- `key.startswith(f"Parent:{pid}")` and `":Child:" not in key`: Finds parents, excluding any children. An alternative is to use a regular expression, such as `r"^Parent:\d+$"`, to match the keys.

All of these queries can be optimized by building indices to restrict the search space to more meaningful subsets.

Let's take a look at how to design an access layer for `shelve`.

Designing an access layer for shelve

Here's how `shelve` might be used by an application. We'll look at parts of an application that edits and saves microblog posts. We'll break the application into two tiers: the application tier and the data tier. Within an application tier, we'll distinguish between two layers:

- **Application processing**: Within the application layer, objects are not persistent. These classes will embody the behavior of the application as a whole. These classes respond to the user selection of commands, menu items, buttons, and other processing elements.
- **Problem domain data model**: These are the objects that will get written to a shelf. These objects embody the state of the application as a whole.

The classes to define an independent `Blog` and `Post` will have to be modified so that we can process them separately in the shelf container. We don't want to create a single, large container object by turning `Blog` into a collection class.

Within the data tier, there might be a number of features, depending on the complexity of the data storage. We'll focus on these two features:

- **Access**: These components provide uniform access to the problem domain objects. We'll focus on the access tier. We'll define an `Access` class that provides access to the `Blog` and `Post` instances. It will also manage the keys to locate the `Blog` and `Post` objects in the shelf.
- **Persistence**: The components serialize and write problem domain objects to persistent storage. This is the `shelve` module. The Access tier will depend on this.

We'll break the `Access` class into three separate pieces. Here's the first part, showing the file open and close operations:

```
import shelve
from typing import cast

class Access:

    def __init__(self) -> None:
        self.database: shelve.Shelf = cast(shelve.Shelf, None)
        self.max: Dict[str, int] = {"Post": 0, "Blog": 0}
```

```
    def new(self, path: Path) -> None:
        self.database: shelve.Shelf = shelve.open(str(path), "n")
        self.max: Dict[str, int] = {"Post": 0, "Blog": 0}
        self.sync()

    def open(self, path: Path) -> None:
        self.database = shelve.open(str(path), "w")
        self.max = self.database["_DB:max"]

    def close(self) -> None:
        if self.database:
            self.database["_DB:max"] = self.max
            self.database.close()
        self.database = cast(shelve.Shelf, None)

    def sync(self) -> None:
        self.database["_DB:max"] = self.max
        self.database.sync()

    def quit(self) -> None:
        self.close()
```

For `Access.new()`, we'll create a new, empty shelf. For `Access.open()`, we'll open an existing shelf. For closing and synchronizing, we've made sure to post a small dictionary of the current maximum key values into the shelf.

The `cast()` function is used to allow us to break the type hint for `self.database` by assigning a `None` object to it. The type hint for this attribute is `shelve.Shelf`. The `cast()` function tells mypy that we're fully aware `None` is not an instance of `Shelf`.

We haven't addressed how to implement a `Save As...` method to make a copy of the file. Nor have we addressed a quit-without-saving option to revert to the previous version of a database file. These additional features involve the use of the `os` module to manage the file copies.

In addition to the basic methods for opening and closing the database, we need methods to perform CRUD operations on blogs and posts. In principle, we would have eight separate methods for this. Here are some of the methods to update the shelf with `Blog` and `Post` objects:

```
def create_blog(self, blog: Blog) -> Blog:
    self.max['Blog'] += 1
    key = f"Blog:{self.max['Blog']}"
    blog._id = key
    self.database[blog._id] = blog
    return blog

def retrieve_blog(self, key: str) -> Blog:
    return self.database[key]

def create_post(self, blog: Blog, post: Post) -> Post:
```

```
        self.max['Post'] += 1
        post_key = f"Post:{self.max['Post']}"
        post._id = post_key
        post._blog_id = blog._id
        self.database[post._id] = post
        return post

    def retrieve_post(self, key: str) -> Post:
        return self.database[key]

    def update_post(self, post: Post) -> Post:
        self.database[post._id] = post
        return post

    def delete_post(self, post: Post) -> None:
        del self.database[post._id]
```

We've provided a minimal set of methods to put `Blog` in the shelf with its associated `Post` instances. When we create a `Blog`, the `create_blog()` method first computes a new key, then updates the `Blog` object with the key, and finally, it persists the `Blog` object in the shelf. We've highlighted the lines that change the shelf contents. Simply setting an item in the shelf, similar to setting an item in a dictionary, will make the object persistent.

When we add a post, we must provide the parent `Blog` so that the two are properly associated on the shelf. In this case, we get the `Blog` key, create a new `Post` key, and then update `Post` with the key values. This updated `Post` can be persisted on the shelf. The highlighted line in `create_post()` makes the object persistent in the shelf.

In the unlikely event that we try to add a `Post` without having previously added the parent `Blog`, we'll have attribute errors because the `Blog._id` attribute will not be available.

We've provided representative methods to replace `Post` and delete `Post`. There are several other possible operations; we didn't include methods to replace `Blog` or delete `Blog`. When we write the method to delete `Blog`, we have to address the question of preventing the deletion when there are still `Post` objects, or cascading the deletion to include `Post` objects. Finally, there are some search methods that act as iterators to query `Blog` and `Post` instances:

```
    def __iter__(self) -> Iterator[Union[Blog, Post]]:
        for k in self.database:
            if k[0] == "_":
                continue  # Skip the administrative objects
            yield self.database[k]
```

```
def blog_iter(self) -> Iterator[Blog]:
    for k in self.database:
        if k.startswith('Blog:'):
            yield self.database[k]

def post_iter(self, blog: Blog) -> Iterator[Post]:
    for k in self.database:
        if k.startswith('Post:'):
            if self.database[k]._blog_id == blog._id:
                yield self.database[k]

def post_title_iter(self, blog: Blog, title: str) -> Iterator[Post]:
    return (p for p in self.post_iter(blog) if p.title == title)
```

We've defined a default iterator, `__iter__()`, to filter out the internal objects that have keys beginning with `_`. So far, we've only defined one such key, `_DB:max`, but this design leaves us with room to invent others.

The `blog_iter()` method iterates through the `Blog` entries. Because the database contains a number of different kinds of objects, we must explicitly discard entries that don't begin with `"Blog:"`. A separate index object is often a better approach than this kind of brute-force filter applied to all keys. We'll look at that in the following *Writing a demonstration script* section.

The `post_iter()` method iterates through posts that are a part of a specific blog. The membership test is based on looking inside each `Post` object to check the `_blog_id` attribute. The `title_iter()` method examines posts that match a particular title. This examines each key in the shelf, which is a potentially inefficient operation.

We also defined an iterator that locates posts that have the requested title in a given blog, `post_title_iter()`. This is a simple generator function that uses the `post_iter()` method function and returns only matching titles.

In the next section, we'll write a demonstration script.

Writing a demonstration script

We'll use a technology spike to show you how an application might use this `Access` class to process the microblog objects. The spike script will save some `Blog` and `Post` objects to a database to show a sequence of operations that an application might use. This demonstration script can be expanded into unit test cases.

More complete unit tests would show us that all the features are present and work correctly. This small spike script shows us how `Access` works:

```
from contextlib import closing
from pathlib import Path

path = Path.cwd() / "data" / "ch11_blog"
with closing(Access()) as access:
    access.new(path)

    # Create Example
    access.create_blog(b1)
    for post in p2, p3:
        access.create_post(b1, post)

    # Retrieve Example
    b = access.retrieve_blog(b1._id)
    print(b._id, b)
    for p in access.post_iter(b):
        print(p._id, p)
```

We've created the `Access` class on the access layer so that it's wrapped in a context manager. The objective is to be sure that the access layer is closed properly, irrespective of any exceptions that might get raised.

With `Access.new()`, we've created a new shelf named `'blog'`. This might be done by a GUI through navigating to File | New. We added the new blog, `b1`, to the shelf. The `Access.create_blog()` method will update the `Blog` object with its shelf key. Perhaps someone filled in the blanks on a page and clicked on New Blog on their GUI application.

Once we've added `Blog`, we can add two posts to it. The key from the parent `Blog` entry will be used to build the keys for each of the child `Post` entries. Again, the idea is that a user filled in some fields and clicked on New Post on the GUI.

There's a final set of queries that dumps the keys and objects from the shelf. This shows us the final outcome of this script. We can perform `Access.retrieve_blog()` to retrieve a blog entry that was created. We can iterate through the posts that are part of that blog using `Access.post_iter()`.

The use of the `contextlib.closing()` context manager ensures that a final `Access.close()` function evaluation will save the database to persistent storage. This will also flush the `self.max` dictionary used to generate unique keys.

The next section talks about how to create indexes to improve efficiency.

Creating indexes to improve efficiency

One of the rules of efficiency is to avoid search. Our previous example of using an iterator over the keys in a shelf is inefficient. To state that more strongly, use of search *defines* an inefficient application. We'll emphasize this.

Brute-force search is perhaps the worst possible way to work with data. Try to design indexes based on subsets or key mappings to improve performance.

To avoid searching, we need to create indexes that list the items users are most likely to want. This saves you reading through the entire shelf to find an item or subset of items. A shelf index can't reference Python objects, as that would change the granularity at which the objects are stored. An index will only list key values, a separate retrieval is done to get the object in question. This makes navigation among objects indirect but still much faster than a brute-force search of all items in the shelf.

As an example of an index, we can keep a list of the `Post` keys associated with each `Blog` in the shelf. We can easily change the `add_blog()`, `add_post()`, and `delete_post()` methods to update the associated `Blog` entry too. Here are the revised versions of these blog update methods:

```
class Access2(Access):

    def create_post(self, blog: Blog, post: Post) -> Post:
        super().create_post(blog, post)
        # Update the index; append doesn't work.
        blog_index = f"_Index:{blog._id}"
        self.database.setdefault(blog_index, [])
        self.database[blog_index] = self.database[blog_index] + [post._id]
        return post

    def delete_post(self, post: Post) -> None:
        super().delete_post(post)
        # Update the index.
        blog_index = f"_Index:{post._blog_id}"
        index_list = self.database[post._blog_id]
        index_list.remove(post._id)
        self.database[post._blog_id] = index_list

    def post_iter(self, blog: Blog) -> Iterator[Post]:
        blog_index = f"_Index:{blog._id}"
        for k in self.database[blog_index]:
```

```
            yield self.database[k]
```

Most of the methods are inherited without change from the `Access` class. We've extended three methods to create a useful index of children for a given blog:

- `create_post()`
- `delete_post()`
- `post_iter()`

The `create_post()` method uses the `create_post()` superclass to save the `Post` object to the shelf. Then it makes sure the `"_Index:{blog}"` object is in the shelf, by using `setdefault()`. This object will be a list with the keys for related posts. The list of keys is updated using the following statement:

```
self.database[blog_index] = self.database[blog_index] + [post._id]
```

This is required to update the shelf. We cannot simply use `self.database[blog_index].append(post._id)`. These kind of *update-in-place* methods of a dictionary don't work as expected with a shelf object. Instead, we must retrieve the object out of the shelf with `self.database[blog_index]`. Update the retrieved object, and then replace the object in the shelf using a simple assignment statement.

Similarly, the `delete_post()` method keeps the index up to date by removing unused posts from `_post_list` of the owning blog. As with `create_post()`, two updates are done to the shelf: a `del` statement removes `Post`, and then the `Blog` object is updated to remove the key from the associated index.

This change alters our queries for a `Post` object in a profound way. We're able to replace a scan of all the items in `post_iter()` with a much more efficient operation. This loop will rapidly yield the `Post` objects based on the keys saved in the `_post_list` attribute of `Blog`. An alternative body is this generator expression:

```
return (self.database[k] for k in blog._post_list)
```

The point of this optimization to the `post_iter()` method is to eliminate the search of *all* the keys for the matching keys. We've replaced searching all keys with a simple iteration over an appropriate sequence of relevant keys. A simple timing test, which alternates between updating `Blog` and `Post` and rendering `Blog` to RST, shows us the following results:

```
Access Layer Access: 33.5 seconds
Access Layer Access2: 4.0 seconds
```

As expected, eliminating the search reduced the time required to process `Blog` and its individual `Posts`. The change is profound: almost 86% of the processing time was wasted in the search for relevant posts.

Let's see how to create a cache.

Creating a cache

We added an index to each `Blog` that locates `Posts` that belong to the `Blog`. We can also add a top-level cache to the shelf that is slightly faster to locate all `Blog` instances. The essential design is similar to what's been shown in the previous section. For each blog to be added or deleted, we must update a cache of valid keys. We must also update the iterators to properly use the index. Here's another class design to mediate the access to our objects:

```
class Access3(Access2):

    def new(self, path: Path) -> None:
        super().new(path)
        self.database["_Index:Blog"] = list()

    def create_blog(self, blog: Blog) -> Blog:
        super().create_blog(blog)
        self.database["_Index:Blog"] += [blog._id]
        return blog

    def blog_iter(self) -> Iterator[Blog]:
        return (self.database[k] for k in
        self.database["_Index:Blog"])
```

When creating a new database, we add an administrative object and an index, with a key of `"_Index:Blog"`. This index will be a list with the keys to each `Blog` entry. When we add a new `Blog` object, we also update this `"_Index:Blog"` object with the revised list of keys.

When we iterate through `Blog` posts, we use the index list instead of a brute-force search of keys in the database. This is slightly faster than using the shelf object's built-in `keys()` method to locate `Blog` posts.

Here are the results as measured:

```
Access Layer Access: 33.5 seconds
Access Layer Access2: 4.0 seconds
Access Layer Access3: 3.9 seconds
```

In the next section, we will learn how to add more index maintenance.

Adding yet more index maintenance

Clearly, the index maintenance aspect of a shelf can grow. With our simple data model, we could easily add more top-level indexes for tags, dates, and titles of `Posts`. Here's another access-layer implementation that defines two indices for `Blogs`. One index simply lists the keys for `Blog` entries. The other index provides keys based on the `Blog` title. We'll assume that the titles are not unique. We'll present this access layer in three parts. Here's the *create* part of the CRUD processing:

```
class Access4(Access3):

    def new(self, path: Path) -> None:
        super().new(path)
        self.database["_Index:Blog_Title"] = dict()

    def create_blog(self, blog):
        super().create_blog(blog)
        blog_title_dict = self.database["_Index:Blog_Title"]
        blog_title_dict.setdefault(blog.title, [])
        blog_title_dict[blog.title].append(blog._id)
        self.database["_Index:Blog_Title"] = blog_title_dict
        return blog
```

We added yet another index. In this example, there is a `dict` that provides us with a list of keys for a given title string. If each title is unique, each of these lists will be a singleton key. If the titles are not unique, each title will have a list of the `Blog` keys.

When we add a `Blog` instance, we also update the title index. The title index requires us to get the existing `dict` from the shelf, append to the list of keys mapped to the `Blog's` title, and then put the `defaultdict` back onto the shelf.

An update to the `Blog` object might involve changing the title of the `Blog` attribute. If there is a title change, this will lead to a complex pair of updates:

1. Remove the old title from the index. Since each title has a list of keys, this operation removes one key from the list. If the list is now empty, the entire title entry can be removed from the dictionary.
2. Add the new title to the index. This echoes the operation shown for adding a new `Blog` object.

Is this additional complexity needed? The only way to be sure is to gather actual performance details for the queries actually used by an application. There is a cost to maintaining an index, and time-saving from using an index to avoid search. There's a fine balance between these, and it often requires some data gathering and experimentation to determine the optimal use of a shelf.

Let's take a look at the writeback alternative to index updates.

The writeback alternative to index updates

We can request that a shelf be opened with `writeback=True`. This will track changes to mutable objects by keeping a cached version of each object in the active memory. This changes the design of the `Access` class. The examples of the `Access` class, shown in the previous sections of this chapter, forced the application to make method calls to `update_blog()` and `update_post()` to be sure a change was persisted to external files. When working in writeback mode, the application is free to mutate an object's values and the `shelf` module will persist the change without any extra method invocations. This automatic update will not update the ancillary indices, however, since they were made by our application's access layer.

In an application where a shelf does not make extensive use of additional index values, the writeback mode can be advantageous. It simplifies application processing, and reduces the need for a sophisticated `Access` class.

Schema evolution

When working with `shelve`, we have to address the problem of *schema evolution*. The class definitions define the schema for the persistent data. The class, however, is not *absolutely* static. If we change a class definition, the schema evolves. How will we fetch objects from the shelf after this change? A good design often involves some combination of the following techniques.

Changes to methods don't change the persisted object representation. We can classify these changes as minor because the shelved data is still compatible with the changed class definition. A new software release can have a new minor version number and users should be confident that it will work without problems.

Changes to attributes will change the persisted object representation. We can call these major changes, and the shelved data will no longer be compatible with the new class definition. Major changes to the representation should not made by *modifying* a class definition. These kinds of changes should be made by adding a new subclass and providing an updated factory function to create instances of either version of the class.

We can be flexible about supporting multiple versions, or we can use one-time conversions. To be flexible, we must rely on factory functions to create instances of objects. A flexible application will avoid creating objects directly. By using a factory function, we're assured that all parts of an application can work consistently. We might do something like this to support flexible schema changes:

```
def make_blog(*args, **kw):
    version = kw.pop('_version',1)
    if version == 1: return Blog(*args, **kw)
    elif version == 2: return Blog2(*args, **kw)
    else: raise ValueError(f"Unknown Version {version}")
```

This kind of factory function requires a `_version` keyword argument to specify which `Blog` class definition to use. This allows us to upgrade a schema to use different classes without breaking our application. The `Access` layer can rely on this kind of function to instantiate correct versions of objects.

An alternative to this level of flexibility is a one-time conversion. This feature of the application will fetch all shelved objects using their old class definition, convert to the new class definition, and store them on a new shelf in the new format.

Summary

We saw the basics of how to use the `shelve` module. This includes creating a shelf and designing keys to access the objects that we've placed in the shelf. We also understood the need for an access layer to perform the lower-level CRUD operations on the shelf. The idea is that we need to distinguish between the class definitions that are focused on our application and other administrative details that support persistence.

Design considerations and tradeoffs

One of the strengths of the `shelve` module is allowing us to persist distinct items very simply. This imposes a design burden to identify the proper granularity, of the items. Too fine a granularity and we waste time assembling a container object from pieces scattered through the database. Too coarse a granularity, and we waste time fetching and storing items that aren't relevant.

Since a shelf requires a key, we must design appropriate keys for our objects. We must also manage the keys for our various objects. This means using additional attributes to store keys, and possibly creating additional collections of keys to act as indices for items on the shelf.

A key used to access an item in a `shelve` database is like a `weakref`; it's an indirect reference. This means that extra processing is required to track and access the items from the reference. For more information on `weakref`, see `Chapter 3`, *Integrating Seamlessly - Basic Special Methods*.

One choice for a key is to locate an attribute or combination of attributes that are proper primary keys and cannot be changed. Another choice is to generate surrogate keys that cannot be changed; this allows all other attributes to be changed. As `shelve` relies on `pickle` to represent the items on the shelf, we have a high-performance native representation of the Python objects. This reduces the complexity of designing classes that will be placed onto a shelf. Any Python object can be persisted.

Application software layers

Because of the relative sophistication available when using `shelve`, our application software must become more properly layered. Generally, we'll look at software architectures with layers such as the following:

- **Presentation layer**: The top-level user interface, either a web presentation or a desktop GUI.
- **Application layer**: The internal services or controllers that make the application work. This could be called the processing model, which is different from the logical data model.
- **Business layer or problem domain model layer**: The objects that define the business domain or problem space. This is sometimes called the logical data model. We've looked at how we might model these objects, using a microblog `Blog` and `Post` example.
- **Infrastructure aspects**: Some applications include a number of cross-cutting concerns or aspects such as logging, security, and network access. These tend to be pervasive and cut across multiple layers.
- **Data access layer**. These are protocols or methods to access data objects. We looked at designing classes to access our application objects from the `shelve` storage.
- **Persistence layer**. This is the physical data model as seen in file storage. The `shelve` module implements persistence.

When looking at this chapter and `Chapter 12`, *Storing and Retrieving Objects with SQLite*, it becomes clear that mastering object-oriented programming involves some higher-level design patterns. We can't simply design classes in isolation; we need to look at how classes are going to be organized into larger structures. Finally, and most importantly, brute-force search is a terrible thing; it must be avoided.

Looking forward

The next chapter will roughly parallel this chapter. We'll look at using SQLite instead of `shelve` for the persistence of our objects. This gets a bit tricky because a SQL database doesn't provide a way to store complex Python objects, leading to the impedance mismatch problem. We'll look at two ways to solve this problem when using a relational database such as SQLite.

In `Chapter 13`, *Transmitting and Sharing Objects*, we'll shift the focus from simple persistence to transmitting and sharing objects. This will rely on the persistence we saw in this chapter, and it will add network protocols into the mix.

Storing and Retrieving Objects via SQLite

There are many applications where we need to persist a large number of distinct objects. The techniques we looked at in `Chapter 10`, *Serializing and Saving - JSON, YAML, Pickle, CSV, and XML*, were biased toward persistence for a single, monolithic object. Sometimes, we need to persist separate, individual objects from a larger domain. For example, we might be saving blog entries, blog posts, authors, and advertisements; each of which must be handled separately.

In `Chapter 11`, *Storing and Retrieving Objects via Shelve*, we looked at storing distinct Python objects in a `shelve` data store. This allowed us to implement the CRUD processing on a domain of individual objects. Each object can be created, retrieved, updated, or deleted without having to load and dump the entire file.

In this chapter, we'll look at mapping Python objects to a relational database; specifically, the `sqlite3` database that is bundled with Python. This will be another example of the **three-tier architecture** design pattern**.**

In this case, the SQLite data tier is a more sophisticated database than `shelve`. SQLite can allow concurrent database updates via locking. SQLite offers an access layer based on the SQL language. It offers persistence by saving SQL tables to the filesystem. Web applications are one example where a database is used instead of simple file persistence to handle concurrent updates to a single pool of data. RESTful data servers, too, frequently use a relational database to provide access to persistent objects.

For scalability, a standalone database server process can be used to isolate all the database transactions. This means persistence can be allocated to a relatively secure host computer, separate from the web application servers and behind appropriate firewalls. MySQL, for example, can be implemented as a standalone server process.

The SQLite3 database that is part of Python is not a standalone database server;

it must be embedded into a host application.

In this chapter, we will cover the following topics:

- SQL databases, persistence, and objects
- Processing application data with SQL
- Mapping Python objects to SQLite BLOB columns
- Mapping Python objects to database rows manually
- Improving performance with indices
- Adding an ORM layer
- Querying pasts given a tag
- Improving performance with indices

Technical requirements

The code files for this chapter are available at `https://git.io/fj2UK`.

SQL databases, persistence, and objects

When using SQLite, our application works with an implicit access layer based on the SQL language. The SQL language is a legacy from an era when object-oriented programming was a rarity. It is heavily biased toward procedural programming. The row-and-column concepts of relational design creates what's termed an *impedance mismatch* with the more complex object model of data. Within SQL databases, we generally focus on three tiers of data modeling, which are shown here:

- **The conceptual model**: These are the entities and relationships implied by the SQL model. These may not be the tables and columns, but may be views of tables and columns. The views may involve selecting rows, projecting columns, or joining multiple tables into the conceptual result. In most cases, the conceptual model can map to Python objects and should correspond with the data model layer of the application tier. This is the place where an **Object-Relational Mapping** (**ORM**) layer is useful.
- **The logical model**: These are the tables, rows, and columns that appear to be in the SQL database. We'll address these entities in our SQL data manipulation statements. We say that these *appear* to exist because the tables and columns are implemented by a physical model that may be somewhat different from the objects defined in the database schema. The results of an SQL query, for example, look table-like, but may not involve storage that parallels the storage of any single, defined table.
- **The physical model**: These are the files, blocks, pages, bits, and bytes of persistent physical storage. These entities are defined by the administrative SQL statements. In some more complex database products, we can exercise some control over the physical model of the data to further tweak the performance. In SQLite, however, we have almost no control over this.

We are confronted with a number of design decisions when using SQL databases. Perhaps the most important one is deciding how to cover the impedance mismatch. How do we handle the mapping between SQL's legacy data model to a Python object model? The following are three common

strategies:

- **Minimal mapping to Python**: This means that we won't build Python objects from the rows retrieved from the database. The application will work entirely within the SQL framework of independent atomic data elements and processing functions. Following this approach tends to avoid a deep emphasis on object-oriented programming. This approach limits us to the four essential SQLite types of `NULL`, `INTEGER`, `REAL`, and `TEXT`, plus the Python additions of `datetime.date` and `datetime.datetime`. While this can be difficult for more complex applications, it is appropriate for certain kinds of scripts when doing database maintenance and support.
- **Manual mapping to Python**: We can define an access layer to map between our Python class definitions and the SQL logical model of tables, columns, rows, and keys. For some specialized purposes, this may be necessary.
- **ORM layer**: We can download and install an ORM layer to handle the mapping between Python objects and the SQL logical model. There are a large number of ORM choices; we'll look at SQLAlchemy as a representative example. An ORM layer is often the simplest and most general approach.

We'll look at all three choices in the following examples. Before we can look at the mappings from SQL to objects, we'll look at the SQL logical model in some detail and cover the no-mapping, pure SQL design strategy first.

The SQL data model – rows and tables

Conceptually, the SQL data model is based on named tables with named columns. A table definition is a flat list of columns with no other structure to the data. Each row is essentially a mutable `@dataclass`. The idea is to imagine the contents within a table as a `list` of individual `@dataclass` objects. The relational model can be described by Python type hints as if it had definitions similar to the following:

```
from dataclasses import dataclass
from typing import Union, Text
import datetime
SQLType = Union[Text, int, float, datetime.datetime, datetime.date, bytes]
@dataclass
class Row:
    column_x: SQLType
    ...
Table = Union[List[Row], Dict[SQLType, Row]]
```

From the type hints, we can see that a database `Table` can be viewed as a list of `Row` instances, or a `Dict` mapping a column to a `Row` instance. The definition of a `Row` is a collection of SQLType column definitions. There is much more to the relational model, but this tiny overview suggests ways we'll use a database in Python. An SQL database has relatively few atomic data types. Generally, the only data structure available is the `Table`, which is a collection of rows. Each `Row` is a simple list of individual column definitions. The data within a `Table` can be used as a simple list of `Row` objects. When we select a column to be a key and the remaining columns to be values, we can consider the table to behave like a dictionary or mapping from key to row value. We didn't clutter up the conceptual type definitions with additional details like multi-column keys, and the nullability of column values.

More complex data structures are built by having rows in one table contain references to a row in another table. These references are built around simply having common values shared by separate tables. When a row's key in one table is referenced by rows in another table, we've effectively defined a hierarchical, one-to-many relationship; this implements a Python nested list or nested

dictionary. We can define many-to-many relationships using an intermediate *association* table, which provides a list of key pairs from each of the associated tables. A set can be built with a table that has a unique, primary key to assure only unique values are collected into the table.

When we define an SQL database, we define a collection of tables and their columns. When we use an SQL database, we manipulate the rows of data collected into the tables.

In the case of SQLite, we have a narrow domain of data types that SQL will process. SQLite handles `NULL`, `INTEGER`, `REAL`, `TEXT`, and `BLOB` data.

The Python types of `None`, `int`, `float`, `str`, and `bytes` are mapped to these SQL types. Similarly, when data of these types is fetched from an SQLite database, the items are converted into Python objects.

The `BLOB` type is a *binary large object*, a collection of bytes of a type defined outside of SQL. This can be used to introduce Python-specific data types into an SQL database. SQLite allows us to add conversion functions for encoding and decoding Python objects into bytes. The `sqlite3` module has already added the `datetime.date` and `datetime.datetime` extensions this way for us. We can add more conversion functions for more specialized processing.

The SQL language can be partitioned into three sublanguages: the **data definition language** (**DDL**), the **data manipulation language** (**DML**), and the **data control language** (**DCL**). The DDL is used to define tables, their columns, and indices. For an example of DDL, we might have some tables defined the following way:

```
CREATE TABLE blog(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT
);
CREATE TABLE post(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    date TIMESTAMP,
    title TEXT,
    rst_text TEXT,
    blog_id INTEGER REFERENCES blog(id)
);
CREATE TABLE tag(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    phrase TEXT UNIQUE ON CONFLICT FAIL
);
CREATE TABLE assoc_post_tag(
```

```
  post_id INTEGER REFERENCES post(id),
  tag_id INTEGER REFERENCES tag(id)
);
```

We've created four tables to represent the `Blog` and `Post` objects for a microblogging application. For more information on the SQL language processed by SQLite, see http://www.sqlite.org/lang.html. For a broader background in SQL, books such as *Creating your MySQL Database: Practical Design Tips and Techniques* will introduce the SQL language in the context of the MySQL database. The SQL language is case insensitive.

For no good reason, we prefer to see SQL keywords in all uppercase to distinguish it from the surrounding Python code.

The `blog` table defines a primary key with the `AUTOINCREMENT` option; this will allow SQLite to assign the key values, saving us from having to generate the keys in our code. The `title` column is the title for a blog. We've defined it to be `TEXT`. In some database products, we must provide a maximum size for a character string. This can help the database engine optimize storage. Since this size is not required in SQLite, so we'll avoid the clutter.

The `post` table defines a primary key as well as date, title, and RST text for the body of the post. Note that we did not reference the tags for a post in this table definition. We'll return to the design patterns required for the following SQL tables. The `post` table does, however, include a formal `REFERENCES` clause to show us that this is a foreign key reference to the owning `blog`.

The `tag` table defines the individual tag text items, and nothing more. The SQL defines the text column as being unique. An attempt to insert a duplicate will cause the transaction to fail.

Finally, we have an association table between `post` and `tag`. This table has only two foreign keys. It associates tags and posts, allowing an unlimited number of tags per post as well as an unlimited number of posts to share a common tag. This association table is a common SQL design pattern to handle this kind of many-to-many relationship. We'll look at some other SQL design patterns in the following section. We can execute the preceding definitions to create our database:

```
import sqlite3
```

```
database = sqlite3.connect('p2_c11_blog.db')
database.executescript(sql_ddl)
```

All database access requires a connection, created with the module function, `sqlite3.connect()`. We provided the name of the file to assign to our database.

The DB-API standard for Python database modules is defined by PEP 249, available at https://www.python.org/dev/peps/pep-0249/. This standard presumes that there is a separate database server process to which our application process will connect. In the case of SQLite, there isn't really a separate process. A `connect()` function is used, however, to comply with the standard. The `sql_ddl` variable in the previous example is simply a long string variable with the four `CREATE TABLE` statements. If there are no error messages, then it means that the table structures have been defined.

The `executescript()` method of a `Connection` object is described in the Python standard library as a *nonstandard shortcut*. Technically, database operations must involve creating a `cursor` object. The following is a standardized approach:

```
from contextlib import closing
with closing(database.cursor()) as cursor:
    for stmt in sql_ddl.split(";"):
        cursor.execute(stmt)
```

While conceptually helpful, this isn't practical. It doesn't work well when there are `";"` characters in comments or text literals. It's better to use the handy `executescript()` shortcut. If we were concerned about the portability to other databases, we'd shift focus to a more strict compliance with the Python DB-API specification. We'll return to the nature of a cursor object in the following section, when looking at queries.

In the next section, we'll discuss CRUD processing via SQL DML statements.

CRUD processing via SQL DML statements

The following four canonical CRUD operations map directly to SQL language statements:

- Creation is done via the `INSERT` statement.
- Retrieval is done via the `SELECT` statement.
- Update is done via the `UPDATE` statement. Some SQL dialects also include a `REPLACE` statement.
- Deletion is done via the `DELETE` statement.

We have to note that we'll often look at literal SQL syntax with all values supplied. This is separate from SQL syntax with binding variable placeholders instead of literal values. The literal SQL syntax is acceptable for scripts; it is perfectly awful for application programming. Building literal SQL statements in an application involves endless string manipulation and famous security problems. See the XKCD comic at http://xkcd.com/327/ for a specific security issue with assembling literal SQL from user-provided text. We'll focus exclusively on SQL with binding variables.

Literal SQL is widely used for scripts. Building literal SQL with user-supplied text is a mistake.

TIP *Never build literal SQL DML statements with string manipulation. It is very high risk to attempt to sanitize user-supplied text.*

The Python DB-API interface defines several ways to bind application variables into SQL statements. SQLite can use positional bindings with `?` or named bindings with `:name`. We'll show you both styles of binding variables. We use an `INSERT` statement to create a new `BLOG` row as shown in the following code snippet:

```
create_blog = """
    INSERT INTO blog(title) VALUES(?)
"""
with closing(database.cursor()) as cursor:
    cursor.execute(create_blog, ("Travel Blog",))
database.commit()
```

We created an SQL statement with a positional bind variable, `?`, for the `title` column of the `blog` table. A cursor object is used to execute the statement after binding a tuple of values. There's only one bind variable, so there's only one value in the tuple. Once the statement has been executed, we have a row in the database. The final commit makes this change persistent, releasing any locks that were held.

We show the SQL statements clearly separated from the surrounding Python code in triple-quoted long string literals. In some applications, the SQL is stored as a separate configuration item. Keeping SQL separate is best handled as a mapping from a statement name to the SQL text. This can simplify application maintenance by keeping the SQL out of the Python programming.

The `DELETE` and `UPDATE` statements require a `WHERE` clause to specify which rows will be changed or removed. To change a blog's title, we might do something like the following:

```
update_blog = """
    UPDATE blog SET title=:new_title WHERE title=:old_title
"""
with closing(database.cursor()) as cursor:
    cursor.execute(
        update_blog,
        dict(
            new_title="2013-2014 Travel",
            old_title="Travel Blog"
        )
    )
database.commit()
```

The `UPDATE` statement shown here has two named bind variables: `:new_title` and `:old_title`. This transaction will update all the rows in the `blog` table that have the given old title, setting the title to the new title. Ideally, the title is unique, and only a single row is touched. SQL operations are defined to work on sets of rows. It's a matter of database design to ensure that a desired row is the content of a set. Hence, the suggestion is to have a unique primary key for every table.

When implementing a delete operation, we always have two choices. We can either prohibit deletes of a parent when children still exist, or we can cascade the deletion of a parent to also delete the relevant children. We'll look at a cascading delete of `blog`, `post`, and `tag` associations. Here's a `DELETE` sequence of statements:

```
delete_post_tag_by_blog_title = """
    DELETE FROM assoc_post_tag
```

```
    WHERE post_id IN (
        SELECT DISTINCT post_id
        FROM blog JOIN post ON blog.id = post.blog_id
        WHERE blog.title=:old_title)
"""
delete_post_by_blog_title = """
    DELETE FROM post WHERE blog_id IN (
        SELECT id FROM BLOG WHERE title=:old_title)
"""
delete_blog_by_title ="""
    DELETE FROM blog WHERE title=:old_title
"""
try:
    with closing(database.cursor()) as cursor:
        title = dict(old_title="2013-2014 Travel")
        cursor.execute("BEGIN")
        cursor.execute(delete_post_tag_by_blog_title, title)
        cursor.execute(delete_post_by_blog_title, title)
        cursor.execute(delete_blog_by_title, title)
    database.commit()
    print("Delete finished normally.")
except Exception as ex:
    print(f"Rollback due to {ex!r}")
    database.rollback()
```

The `DELETE` operation for an entire blog cascades into three separate delete operation. First, we deleted all the rows from `assoc_post_tag` for a given `blog` based on the title. Note the nested query inside the value of `delete_post_tag_by_blog_title`; we'll look at queries in the next section. Navigation among tables is a common issue with SQL construction.

In this case, we have to query the blog-to-post relationship to locate the `post` table keys that will be removed; then, we can remove rows from `assoc_post_tag` for the posts associated with a blog that will be removed. Next, we deleted all the posts belonging to a particular blog. This too involves a nested query to locate the IDs of the blog based on the title. Finally, we can delete the blog itself.

This is an example of an explicit cascade delete design, where we have to cascade the operation from the `blog` table to two other tables. We wrapped the entire suite of delete in a `with` context so that it would all commit as a single transaction. In the event of failure, it would roll back the partial changes, leaving the database as it was.

It seems like this kind of operation would benefit from using the `executescript()` method of the database. The problem with `executescript()` is that all of the values in the SQL statements must be literal; it does not bind values. It works nicely for data definition. It is a poor choice for data manipulation shown here, where the blog title must be bound into each statement.

Let's discuss row querying with the SQL SELECT statement in the next section.

Querying rows with the SQL SELECT statement

It's possible to write a substantial book on the SELECT statement alone. We'll skip all but the most fundamental features of SELECT. Our purpose is to cover just enough SQL to store and retrieve objects from a database.

In most of the previous examples, we've created a cursor to execute SQL statements. For DDL and other DML statements, the presence or absence of a cursor doesn't matter very much. A nonstandard shortcut that eliminates the explicit creation of the cursor and greatly simplifies SQL programming.

For a query, however, the cursor is essential for retrieving the rows from the database. A cursor object maintains the query state so that multiple rows can be fetched. When a cursor is closed, any internal buffers or locks are released. To locate a blog by title, we can start with something as simple as the following code:

```
SELECT * FROM blog WHERE title=?
```

After executing the SQL query, we need to fetch the resulting collection of row objects. Even when we're expecting one row as a response, in the SQL world, everything is a collection. Generally, every result set from a SELECT query looks like a table with rows and columns defined by the SELECT statement.

In this case, using SELECT * means we've avoided enumerating the expected result columns. This might lead to a large number of columns being retrieved. Using an explicit cursor, we'd execute this as follows:

```
query_blog_by_title = """
    SELECT * FROM blog WHERE title=?
"""
with closing(database.cursor()) as blog_cursor:
    blog_cursor.execute(query_blog_by_title, ("2013-2014 Travel",))
    for blog in blog_cursor.fetchall():
        print("Blog", blog)
```

This follows the previous pattern of creating a cursor and executing the SQL

query statement with a bound value. The `fetchall()` method of a cursor will retrieve all of the result rows. This query will process all blogs with the same title.

Here's a common optimization for doing this using the SQLite shortcuts:

```
query_blog_by_title= """
    SELECT * FROM blog WHERE title=?
"""
cursor = database.execute(
    query_blog_by_title, ("2013-2014 Travel",))
for blog in cursor:
    print(blog[0], blog[1])
```

We've bound the requested blog title to the `"?"` parameter in the `SELECT` statement. The result of the `execute()` function will be a cursor object. When used as an iterable, the cursor will yield all the rows in the result set, and close the cursor when the iteration is complete.

This shortened form can be handy for queries or simple transactions. For more complex transactions, like a cascading delete, then an explicit cursor with a `commit()` is required to assure the transaction is either executed completely or not at all. We'll look at the semantics of SQL transactions next.

SQL transactions and the ACID properties

As we've seen, the SQL DML statements map to the CRUD operations. When discussing the features of the SQL transactions, we'll be looking at the sequences of the `INSERT`, `SELECT`, `UPDATE`, and `DELETE` statements.

The ACID properties are *Atomic*, *Consistent*, *Isolated*, and *Durable*. These are essential features of a transaction that consists of multiple database operations. For more information, see `Chapter 11`, *Storing and Retrieving Objects via Shelve*.

All the SQL DML statements operate within the context of an SQL transaction. The entire transaction must be committed as a whole, or rolled back as a whole. This supports the atomicity property by creating a single, atomic, indivisible change from one consistent state to the next.

SQL DDL statements (that is, `CREATE`, `DROP`) do not work within a transaction. They implicitly end any previous in-process transaction. After all, they're changing the structure of the database; they're a different kind of statement, and the transaction concept doesn't really apply. Each SQL database product varies slightly in whether or not a commit is necessary when creating tables or indices. Issuing a `database.commit()` can't hurt; for some database engines, it may be required.

Unless working in a special **read uncommitted** mode, each connection to the database sees a consistent version of the data containing only the results of the committed transactions. Uncommitted transactions are generally invisible to other database client processes, supporting the consistency property.

An SQL transaction also supports the isolation property. SQLite supports several different **isolation level** settings. The isolation level defines how the SQL DML statements interact among multiple, concurrent processes. This is based on how locks are used and how a process' SQL requests are delayed waiting for locks. From Python, the isolation level is set when the connection is made to the database.

Each SQL database product takes a different approach to the isolation level and locking. There's no single model.

In the case of SQLite, there are four isolation levels that define the locking and the nature of transactions. For details, see http://www.sqlite.org/isolation.html.

The following bullet list talks about these isolation levels:

- `isolation_level=None`: This is the default, otherwise known as the **autocommit** mode. In this mode, each individual SQL statement is committed to the database as it's executed. This can break the atomicity of complex transactions. For a data warehouse query application, however, it's ideal.
- `isolation_level='DEFERRED'`: In this mode, locks are acquired late in the transaction. The `BEGIN` statement, for example, does not immediately acquire any locks. Other read operations (examples include the `SELECT` statements) will acquire shared locks. Write operations will acquire reserved locks. While this can maximize the concurrency, it can also lead to deadlocks among competing transaction processes.
- `isolation_level='IMMEDIATE'`: In this mode, the transaction `BEGIN` statement acquires a lock that prevents all writes. Reads, however, will continue normally. This avoids deadlocks, and works well when transactions can be completed quickly.
- `isolation_level='EXCLUSIVE'`: In this mode, the transaction `BEGIN` statement acquires a lock that prevents all access except for connections in a special read uncommitted mode.

The durability property is guaranteed for all committed transactions. The data is written to the database storage.

The SQL rules require us to execute the `BEGIN TRANSACTION` and `COMMIT TRANSACTION` statements to bracket a sequence of steps. In the event of an error, a `ROLLBACK TRANSACTION` statement is required to unwind the potential changes. The Python interface simplifies this. We can execute an SQL `BEGIN` statement. The other statements are provided as functions of the `sqlite3.Connection` object; we don't execute SQL statements to end a transaction. We might write things like the following code to be explicit:

```
database = sqlite3.connect('p2_c11_blog.db', isolation_level='DEFERRED')
try:
    with closing(database.cursor()) as cursor:
```

```
        cursor.execute("BEGIN")
        # cursor.execute("some statement")
        # cursor.execute("another statement")
    database.commit()
except Exception as e:
    database.rollback()
```

We selected an isolation level of `DEFERRED` when we made the database connection. This leads to a requirement that we explicitly begin and end each transaction. One typical scenario is to wrap the relevant DML in a `try` block and commit the transaction if things worked, or roll back the transaction in the case of a problem. We can simplify this by using the `sqlite3.Connection` object as a context manager as follows:

```
database = sqlite3.connect('p2_c11_blog.db', isolation_level='DEFERRED')
with database:
    database.execute("some statement")
    database.execute("another statement")
```

This is similar to the previous example. We opened the database in the same way. Rather than executing an explicit `BEGIN` statement, we entered a context; the context executes the `BEGIN` statement for us.

At the end of the `with` context, `database.commit()` will be done automatically. In the event of an exception, a `database.rollback()` will be done, and the exception will be raised by the `with` statement.

This kind of shorthand is helpful in a web server where a connection only needs to last for the duration of a single web request. When doing queries, the isolation level can be left as the default, resulting in a very quick access to data.

In the next section, we'll design primary and foreign database keys.

Designing primary and foreign database keys

SQL tables don't require a primary key. However, it's a rather poor design that omits primary keys for the rows of a given table. As we noted in `Chapter 11`, *Storing and Retrieving Objects via Shelve*, there might be an attribute (or a combination of attributes) that makes a proper primary key. It's also entirely possible that no attribute is suitable as a primary key and we must define surrogate keys.

The previous examples use surrogate keys created by SQLite. This is perhaps the simplest kind of design because it imposes the fewest constraints on the data. If we don't generate surrogate keys, then a primary key value cannot be updated; this becomes a rule that the application programming must enforce. This creates a complex edge case if we need to correct an error in the primary key value. One way to do this is to drop and recreate the constraints. Another way to do this is to delete the faulty row and reinsert the row with the corrected key.

When there are cascading deletes, then the transaction required to correct a primary key can become very complex. Using a surrogate key prevents these kinds of problems. All relationships among tables are done via the primary keys and foreign key references. There are two extremely common design patterns for relationships. The preceding tables show us these two principle design patterns. There are three design patterns for relationships, as follows:

- **One-to-many**: A number of children belong to a single parent object. For example, this is the relationship between one parent blog and many child posts. The `REFERENCES` clause shows us that many rows in the `post` table will reference one row from the `blog` table.
- **Many-to-many**: This relationship is between many posts and many tags. This requires an intermediate association table between the `post` and `tag` tables; the intermediate table has two (or more) foreign key associations. The many-to-many association table can also have attributes of its own.
- **One-to-one**: This relationship is a less common design pattern. There's no technical difference from a one-to-many relationship. This is a question of

the cardinality of rows on the child side of the relationship. To enforce a one-to-one relationship, some processing must prevent the creation of additional children.

In a database design, there are several kinds of constraints on a relationship: the relationship might be described as optional or mandatory; there might be cardinality limits on the relationship. Sometimes, these optionality and cardinality constraints are summarized with short descriptions such as `0:m` meaning *zero to many* or *optional one to many*. The optionality and cardinality constraints are often an important part of the application programming logic; there are no formal ways to state these constraints in the SQLite database. The essential table relationships can be implemented in the database in either or both of the following ways:

- **Explicit**: We could call these declared, as they're part of the DDL declaration for a database. Ideally, they're enforced by the database server, and failure to comply with the relationship's constraints can lead to an error of some kind. These relationships will also be repeated in queries.
- **Implicit**: These are relationships that are stated only in queries; they are not a formal part of the DDL.

Note that our table definitions implemented a one-to-many relationship between a blog and the various entries within that blog. We've made use of these relationships in the various queries that we wrote.

In the next section, we'll see how to process application data with SQL.

Processing application data with SQL

The examples in the previous sections show us what we can call **procedural** SQL processing. We've eschewed any object-oriented design from our problem domain objects. Rather than working with the `Blog` and `Post` objects, we're working with the data elements that SQLite can process string, date, float, and integer values. We've used mostly procedural-style programming.

We can see that a series of queries can be done to locate a blog, all posts that are part of the blog, and all tags that are associated with a post associated with a blog. The processing to retrieve a complete blog, all posts, and all tags on the posts would look like the following code:

```
query_blog_by_title = """
    SELECT * FROM blog WHERE title=?
"""
query_post_by_blog_id = """
    SELECT * FROM post WHERE blog_id=?
"""
query_tag_by_post_id = """
    SELECT tag.*
    FROM tag
    JOIN assoc_post_tag ON tag.id = assoc_post_tag.tag_id
    WHERE assoc_post_tag.post_id=?
"""
with closing(database.cursor()) as blog_cursor:
    blog_cursor.execute(query_blog_by_title, ("2013-2014 Travel",))
    for blog in blog_cursor.fetchall():
        print("Blog", blog)
        with closing(database.cursor()) as post_cursor:
            post_cursor.execute(query_post_by_blog_id, (blog[0],))
            for post in post_cursor:
                print("Post", post)
                with closing(database.cursor()) as tag_cursor:
                    tag_cursor.execute(query_tag_by_post_id, (post[0],))
                    for tag in tag_cursor.fetchall():
                        print("Tag", tag)
```

We defined three SQL queries. The first fetches the blogs by the title. For each blog, we fetched all the posts that belong to this blog. Finally, we fetched all tags that are associated with a given post.

The second query implicitly repeats the `REFERENCES` definition between the `post` table and the `blog` table. We're finding child posts of a specific blog parent; we need to repeat some of the table definitions during the query.

The third query involves a relational join between rows of the `assoc_post_tag` table and the `tag` table. The `JOIN` clause recapitulates the foreign key reference in the table definitions. The `WHERE` clause also repeats a `REFERENCES` clause in the table definitions.

Because multiple tables were joined in the third query, using `SELECT *` will produce columns from all of the tables. We're really only interested in attributes of the `TAG` table, so we use `SELECT TAG.*` to produce only the desired columns.

These queries provide us with all of the individual bits and pieces of the data. However, these queries don't reconstruct Python objects for us. If we have more complex class definitions, we have to build objects from the individual pieces of data that we retrieved. In particular, if our Python class definitions have important method functions, we'll need a better SQL-to-Python mapping to make use of more complete Python class definitions.

In the next section, we'll see how to implement class-like processing in pure SQL.

Implementing class-like processing in pure SQL

Let's look at a somewhat more complex definition of a `Blog` class. This definition is repeated from `Chapter 10`, *Serializing and Saving - JSON, YAML, Pickle, CSV, and XML*, as well as `Chapter 11`, *Storing and Retrieving Objects via Shelve*. This definition is as follows:

```
from dataclasses import dataclass, field, asdict

@dataclass
class Blog:

    title: str
    underline: str = field(init=False)

    # Part of the persistence, not essential to the class.
    _id: str = field(default="", init=False, compare=False)

    def __post_init__(self) -> None:
        self.underline = "=" * len(self.title)
```

This dataclass provides the essential title attribute for a blog. It has an optional attribute with the internal database ID assigned to the blog entry.

Here's the start of an `Access` class to retrieve `Blog` and `Post` objects:

```
class Access:

    def open(self, path: Path) -> None:
        self.database = sqlite3.connect(path)
        self.database.row_factory = sqlite3.Row

    def get_blog(self, id: str) -> Blog:
        query_blog = """
            SELECT * FROM blog WHERE id=?
        """
        row = self.database.execute(query_blog, (id,)).fetchone()
        blog = Blog(title=row["TITLE"])
        blog._id = row["ID"]
        return blog
```

This `Access` class has a method that creates a `Blog` object from the columns in the relational database. The `__post_init__()` method will create the expected `underline` attribute value. This shows the essential technique for working from relational data to an object.

Retrieving the `Post` instances associated with a `Blog` instance isn't trivial. It requires using an `access` object to fetch more rows from the database. It can't be done directly via some kind of attribute defined solely within the `Blog` class. We'll look at this in more depth when we look at the access layers and **object-relational management** (**ORM**).

Mapping Python objects to SQLite BLOB columns

We can map SQL columns to class definitions so that we can create proper Python object instances from data in a database. SQLite includes a **Binary Large Object** (**BLOB**) data type. We can pickle our Python objects and store them in the BLOB columns. We can work out a string representation of our Python objects (for example, using the JSON or YAML notation) and use SQLite text columns too.

This technique must be used cautiously because it effectively defeats SQL processing. A BLOB column cannot be used for SQL DML operations. We can't index it or use it in the search criteria of DML statements.

SQLite BLOB mapping should be reserved for objects where it's acceptable to be opaque to the surrounding SQL processing. The most common examples are media objects such as videos, still images, or sound clips. SQL is biased towards text and numeric fields. It doesn't generally handle more complex objects.

If we're working with financial data, our application should use the `decimal.Decimal` values. . The reasons were discussed in `Chapter 8`, *Creating Numbers*; currency computations will be incorrect when performed with float values. We might want to query or calculate in SQL using this kind of data. As `decimal.Decimal` is not directly supported by SQLite, we need to extend SQLite to handle values of this type.

There are two directions to this: conversion and adaptation. We need to **adapt** Python data to SQLite, and we need to **convert** SQLite data back to Python. Here are two functions and the requests to register them:

```
import decimal
def adapt_currency(value):
    return str(value)
sqlite3.register_adapter(decimal.Decimal, adapt_currency)
def convert_currency(bytes):
    return decimal.Decimal(bytes.decode())
sqlite3.register_converter("DECIMAL", convert_currency)
```

We've written an `adapt_currency()` function that will adapt `decimal.Decimal` objects into a suitable form for the database. In this case, we've done nothing more than a simple conversion to a string. We've registered the adapter function so that SQLite's interface can convert objects of class `decimal.Decimal` using the registered adapter function. We've also written a `convert_currency()` function that will convert SQLite bytes objects into the Python `decimal.Decimal` objects. We've registered the `converter` function so that columns of the `DECIMAL` type will be properly converted to Python objects.

Once we've defined the adapters and converters, we can use `DECIMAL` as a fully supported column type. For this to work properly, we must inform SQLite by setting `detect_types=sqlite3.PARSE_DECLTYPES` when making the database connection. Here's a table definition that uses our new column data type:

```
CREATE TABLE budget(
    year INTEGER,
    month INTEGER,
    category TEXT,
    amount DECIMAL
)
```

As with text fields, a maximum size isn't required by SQLite. Other database products require a size to optimize storage and performance. We can use our new column type definition like this:

```
database = sqlite3.connect('p2_c11_blog.db', detect_types=sqlite3.PARSE_DECLTYPES)
database.execute(decimal_ddl)

insert_budget= """
    INSERT INTO budget(year, month, category, amount)
    VALUES(:year, :month, :category, :amount)
"""
database.execute(insert_budget,
    dict(year=2013, month=1, category="fuel", amount=decimal.Decimal('256.78')))
database.execute(insert_budget,
    dict(year=2013, month=2, category="fuel", amount=decimal.Decimal('287.65')))

query_budget= """
    SELECT * FROM budget
"""
for row in database.execute(query_budget):
    print(row)
```

We created a database connection that requires declared types to be mapped via a converter function. Once we have the connection, we can create our table using a new `DECIMAL` column type.

When we insert rows into the table, we use proper `decimal.Decimal` objects. When

we fetch rows from the table, we'll see that we get proper `decimal.Decimal` objects back from the database. The following is the output:

```
(2013, 1, 'fuel', Decimal('256.78'))
(2013, 2, 'fuel', Decimal('287.65'))
```

This shows us that our `decimal.Decimal` objects were properly stored and recovered from the database. We can write adapters and converters for any Python class. To do this, we need to invent a proper byte representation of the object. As a string is so easily transformed into bytes, creating and parsing strings is often the simplest way to proceed. Bytes can be created from strings using the `encode()` method of a string. Similarly, strings can be recovered from bytes using the bytes `decode()` method.

Let's take a look at how to map Python objects to database rows manually.

Mapping Python objects to database rows manually

We can map SQL rows to class definitions so that we can create proper Python object instances from the data in a database. If we're careful with our database and class definitions, this isn't impossibly complex. If, however, we're careless, we can create Python objects where the SQL representation is quite complex. One consequence of the complexity is that numerous queries are involved in mapping between object and database rows. The challenge is to strike a balance between object-oriented design and the constraints imposed by the SQL database.

We will have to modify our class definitions to be more aware of the SQL implementation. We'll make several modifications to the `Blog` and `Post` class designs shown in `Chapter 11`, *Storing and Retrieving Objects via Shelve*.

Here's a `Blog` class definition:

```
@dataclass
class Blog:

    title: str
    underline: str = field(init=False)

    # Part of the persistence, not essential to the class.
    _id: str = field(default="", init=False, compare=False)
    _access: Optional[ref] = field(init=False, repr=False, default=None, compare=False)

    def __post_init__(self) -> None:
        self.underline = "=" * len(self.title)

    @property
    def entries(self) -> List['Post']:
        if self._access and self._access():
            posts = cast('Access', self._access()).post_iter(self)
            return list(posts)
        raise RuntimeError("Can't work with Blog: no associated Access instance")

    def by_tag(self) -> Dict[str, List[Dict[str, Any]]]:
        if self._access and self._access():
            return cast('Access', self._access()).post_by_tag(self)
        raise RuntimeError("Can't work with Blog: no associated Access instance")
```

The core elements of a `Blog` instance are present: the `title` and a `property` that

provides a list of `Post` entries. This precisely matches the feature set shown in previous chapters.

In order to work with the persistence of objects, it's convenient to include the database key value. In this definition, it's the `_id` field. This field is not part of the initialization, nor is it part of the comparison for instances of the `Blog` class. The underline attribute is computed, as in previous examples.

The `entries` property relies on the optional `_access` field. This provides a reference to the `Access` class that provides a database connection and handles the SQL mappings from the database tables. This value is injected by the `Access` class when an object is retrieved.

The list of `Post` objects requires a definition of the required class. The `Post` class definition is as follows:

```
@dataclass
class Post:

    date: datetime.datetime
    title: str
    rst_text: str
    tags: List[str] = field(default_factory=list)
    _id: str = field(default="", init=False, compare=False)

    def append(self, tag):
        self.tags.append(tag)
```

The fields describe an individual `Post`; these fields include the date, the title, the body in RST notation, and a list of tags to further categorize posts. As with `Blog`, we've allowed for a database `_id` field as a first-class part of the object.

Once we have these class definitions, we can write an access layer that moves data between objects of these classes and the database. The access layer implements a more complex version of converting and adapting Python classes to rows of a table in the database.

Let's see how to design an access layer for SQLite.

Designing an access layer for SQLite

For this small object model, we can implement the entire access layer in a single class. This class will include methods to perform CRUD operations on each of our persistent classes.

This example won't painstakingly include all of the methods for a complete access layer. We'll show you the important ones. We'll break this down into several sections to deal with `Blogs`, `Posts`, and iterators. Here's the first part of our access layer:

```
# An access layer to map back and forth between Python objects and SQL rows.
class Access:
    get_last_id = """
        SELECT last_insert_rowid()
    """

    def open(self, path: Path) -> None:
        self.database = sqlite3.connect(path)
        self.database.row_factory = sqlite3.Row

    def get_blog(self, id: str) -> Blog:
        query_blog = """
            SELECT * FROM blog WHERE id=?
        """
        row = self.database.execute(query_blog, (id,)).fetchone()
        blog = Blog(title=row["TITLE"])
        blog._id = row["ID"]
        blog._access = ref(self)
        return blog

    def add_blog(self, blog: Blog) -> Blog:
        insert_blog = """
            INSERT INTO blog(title) VALUES(:title)
        """
        self.database.execute(insert_blog, dict(title=blog.title))
        row = self.database.execute(self.get_last_id).fetchone()
        blog._id = str(row[0])
        blog._access = ref(self)
        return blog
```

In addition to opening the database, the `open()` method sets `Connection.row_factory` to use the `sqlite3.Row` class instead of a simple tuple. The `Row` class allows access via the numeric index, as well as the column name.

The `get_blog()` method constructs a `Blog` object from the database row that is fetched. Because we're using the `sqlite3.Row` object, we can refer to columns by name. This clarifies the mapping between SQL and the Python class. The two

additional attributes, `_id` and `_access` must be set separately; they're part of the access layer, and not central to the problem domain.

The `add_blog()` method inserts a row into the `blog` table based on the value of a `Blog` object. This is a three-step operation. First, we create the new row. Then, we perform an SQL query to get the row key that was assigned to the row. Finally, the original blog instance is updated with the assigned database key and a reference to the `Access` instance.

Note that our table definitions use `INTEGER PRIMARY KEY AUTOINCREMENT`. Because of this, the table's primary key is the `_id` attribute of the row, and the assigned row key will be available through the `last_insert_rowid()` function. This allows us to retrieve the row key that was created by SQLite; we can then put this into the Python object for future reference. Here's how we can retrieve an individual `Post` object from the database:

```
def get_post(self, id: str) -> Post:
    query_post = """
        SELECT * FROM post WHERE id=?
    """
    row = self.database.execute(query_post, (id,)).fetchone()
    post = Post(
        title=row["TITLE"], date=row["DATE"], rst_text=row["RST_TEXT"]
    )
    post._id = row["ID"]
    # Get tag text, too
    query_tags = """
        SELECT tag.*
        FROM tag JOIN assoc_post_tag ON tag.id = assoc_post_tag.tag_id
        WHERE assoc_post_tag.post_id=?
    """
    results = self.database.execute(query_tags, (id,))
    for tag_id, phrase in results:
        post.append(phrase)
    return post
```

To build `Post`, we have two queries: first, we fetch a row from the `post` table to build part of the `Post` object. This includes injecting the database ID into the resulting instance. Then, we fetch the association rows joined with the rows from the `tag` table. This is used to build the tag list for the `Post` object.

When we save a `Post` object, it will also have several parts. A row must be added to the `POST` table. Additionally, rows need to be added to the `assoc_post_tag` table. If a tag is new, then a row might need to be added to the `tag` table. If the tag exists, then we simply associate the post with an existing tag's key. Here's the `add_post()` method function:

```
def add_post(self, blog: Blog, post: Post) -> Post:
    insert_post = """
        INSERT INTO post(title, date, rst_text, blog_id) VALUES(:title, :date, :rst_text
    """
    query_tag = """
        SELECT * FROM tag WHERE phrase=?
    """
    insert_tag = """
        INSERT INTO tag(phrase) VALUES(?)
    """
    insert_association = """
        INSERT INTO assoc_post_tag(post_id, tag_id) VALUES(:post_id, :tag_id)
    """
    try:
        with closing(self.database.cursor()) as cursor:
            cursor.execute(
                insert_post,
                dict(
                    title=post.title,
                    date=post.date,
                    rst_text=post.rst_text,
                    blog_id=blog._id,
                ),
            )
            row = cursor.execute(self.get_last_id).fetchone()
            post._id = str(row[0])
            for tag in post.tags:
                tag_row = cursor.execute(query_tag, (tag,)).fetchone()
                if tag_row is not None:
                    tag_id = tag_row["ID"]
                else:
                    cursor.execute(insert_tag, (tag,))
                    row = cursor.execute(self.get_last_id).fetchone()
                    tag_id = str(row[0])
                cursor.execute(
                    insert_association,
                    dict(tag_id=tag_id, post_id=post._id)
                )
        self.database.commit()
    except Exception as ex:
        self.database.rollback()
        raise
    return post
```

The process of creating a complete post in the database involves several SQL steps. We've used the `insert_post` statement to create the row in the `post` table. We'll also use the generic `get_last_id` query to return the assigned primary key for the new `post` row.

The `query_tag` statement is used to determine whether the tag exists in the database or not. If the result of the query is not `None`, it means that a `tag` row was found, and we have the ID for that row. Otherwise, the `insert_tag` statement must be used to create a row; the `get_last_id` query must be used to determine the assigned key.

Each `post` is associated with the relevant tags by inserting rows into the `assoc_post_tag` table. The `insert_association` statement creates the necessary row. Here are two iterator-style queries to locate `Blogs` and `Posts`:

```
def blog_iter(self) -> Iterator[Blog]:
    query = """
        SELECT * FROM blog
    """
    results = self.database.execute(query)
    for row in results:
        blog = Blog(title=row["TITLE"])
        blog._id = row["ID"]
        blog._access = ref(self)
        yield blog

def post_iter(self, blog: Blog) -> Iterator[Post]:
    query = """
        SELECT id FROM post WHERE blog_id=?
    """
    results = self.database.execute(query, (blog._id,))
    for row in results:
        yield self.get_post(row["ID"])
```

The `blog_iter()` method function locates all the `BLOG` rows and builds `Blog` instances from the rows. The `post_iter()` method function locates `POST` IDs that are associated with a `BLOG` ID. The `POST` IDs are used with the `get_post()` method to build the `Post` instances. As `get_post()` will perform another query against the `POST` table, there's an optimization possible between these two methods.

Let's see how to implement container relationships in the next section.

Implementing container relationships

When we looked at `Blog` objects in `Chapter 11`, *Storing and Retrieving Objects via Shelve*, we defined a `by_tag()` method to emit useful dictionary representations of the posts organized by the relevant tag strings. The method had a definition with a type hint that was `Dict[str, List[Dict[str, Any]]]`. Because it provided information useful for creating tags with links, the `by_tag()` method was a helpful part of rendering a `Blog` instance as RST or HTML. Additionally, the `entries` property returns the complete collection of `Post` instances attached to the `Blog`.

Ideally, the application model class definitions such as `Blog` and `Post` are utterly divorced from the access layer objects, which persist them in the external storage. These two use cases suggest the model layer objects must have references to the access layer. There are several strategies, including the following:

- A global `Access` object is used by client classes to perform these query operations. This breaks the encapsulation idea: a `Blog` is no longer a container for `Post` entries. Instead, an `Access` object is a container for both. The class definitions are simplified. All other processing is made more complex.

- Include a reference to an access layer object within each `Blog` object, allowing a client class to work with a `Blog` object unaware of an access layer. This makes the model layer class definitions a bit more complex. It makes the client work somewhat simpler. The advantage of this technique is it makes the model layer objects behave more like complex Python objects. A complex object may be forced to fetch child objects from the database, but if this can be done transparently, it makes the overall application somewhat simpler.

As a concrete example, we'll add a `Blog.by_tag()` feature. The idea is to return a dictionary of tags and post information as a complex dictionary. This requires considerable work by an access layer object to locate and fetch the dictionary representations of `Post` instances.

There's no trivial mapping from the relational columns to objects. Therefore, an `Access` class must build each class of object. As an example, the method for fetching a `Blog` instance is shown as follows:

```
def get_blog(self, id: str) -> Blog:
    query_blog = """
        SELECT * FROM blog WHERE id=?
    """
    row = self.database.execute(query_blog, (id,)).fetchone()
    blog = Blog(id=row["ID"], title=row["TITLE"])
    blog._access = ref(self)
    return blog
```

The relational query retrieves the various attributes for recreating a `Blog` instance. In addition to creating the core fields, each `Blog` object has an optional `_access` attribute. This is not provided at initialization, nor is it part of the representation or comparison of `Blog` objects. The value is a weak reference to an instance of an `Access` class. This object will embody the rather complex SQL query required to do the work. This association is inserted by the access object each time a blog instance is retrieved.

Associating blogs, posts, and tags will require a rather complex SQL query. Here's the `SELECT` statement required to traverse the associations and locate tag phrases and associated post identifiers:

```
query_by_tag = """
    SELECT tag.phrase, post.id
    FROM tag
    JOIN assoc_post_tag ON tag.id = assoc_post_tag.tag_id
    JOIN post ON post.id = assoc_post_tag.post_id
    JOIN blog ON post.blog_id = blog.id
    WHERE blog.title=?
"""
```

This query's result set is a table-like sequence of rows with two attributes: `tag.phrase`, `post.id`. The `SELECT` statement defines three join operations between the `blog`, `post`, tag, and `assoc_post_tag` tables. The rules are provided via the `ON` clauses within each `JOIN` specification. The `tag` table is joined with the `assoc_post_tag` table by matching values of the `tag.id` column with values of the `assoc_post_tag.tag_id` column. Similarly, the `post` table is associated by matching values of the `post.id` column with values of the `assoc_post_tag.post_id` column. Additionally, the `blog` table is associated by matching values of the `post.blog_id` column with values of the `blog.id` column.

The `by_tag()` method of the `Access` class using this query is as follows:

```
def post_by_tag(self, blog: Blog) -> Dict[str, List[Dict[str, Any]]]:
    results = self.database.execute(
        self.query_by_tag, (blog.title,))
    tags: DefaultDict[str, List[Dict[str, Any]]] = defaultdict(list)
    for phrase, post_id in results.fetchall():
        tags[phrase].append(asdict(self.get_post(post_id)))
    return tags
```

It's important to note this complex SQL query is dissociated from the table definitions. SQL is not an object-oriented programming language. There's no tidy class to bundle data and processing together. Using procedural programming with SQL like this tends to break the object model.

The `by_tag()` method of the `blog` class uses this method of the `Access` class to do the actual fetches of the various posts. A client application can then do the following:

```
blog = some_access_object.get_blog(id=1)
tag_link = blog.by_tag()
```

The returned `Blog` instance behaves as if it was a simple Python class definition with a simple collection of `Post` instances. To make this happen, we require some careful injection of access layer references into the returned objects.

In the next section, we'll see how to improve the performance with indices.

Improving performance with indices

One of the ways to improve the performance of a relational database such as SQLite is to make join operations faster. The ideal way to do this is to include enough index information that slow search operations aren't done to find matching rows.

When we define a column that might be used in a query, we should consider building an index for that column. This means adding yet more SQL DDL statements to our table definitions.

An index is a separate storage but is tied to a specific table and column. The SQL looks like the following code:

```
CREATE INDEX ix_blog_title ON blog(title);
```

This will create an index on the `title` column of the `blog` table. Nothing else needs to be done. The SQL database will use the index when performing queries based on the indexed column. When data is created, updated, or deleted, the index will be adjusted automatically.

Indexes involve storage and computational overheads. An index that's rarely used might be so costly to create and maintain that it becomes a performance hindrance rather than a help. On the other hand, some indexes are so important that they can have spectacular performance improvements. In all cases, we don't have direct control over the database algorithms being used; the best we can do is create the index and measure the performance's impact.

In some database products, defining a column to be a key might automatically include having an index added. In other cases, it's the presence of an index that forces a column to be considered as a unique key. The rules for this are usually stated quite clearly in the database's DDL section. The documentation for SQLite, for example, says this:

In most cases, UNIQUE and PRIMARY KEY constraints are implemented by creating a unique index in the database.

It goes on to list two exceptions. One of these, the integer primary key exception, is the design pattern we've been using to force the database to create surrogate keys for us. Therefore, our integer primary key design will not create any additional indices.

In the next section, we'll discuss adding an ORM layer.

Adding an ORM layer

There are a fairly large number of Python ORM projects. A list of these can be found at `https://wiki.python.org/moin/HigherLevelDatabaseProgramming`.

We're going to pick just one of these as an example. We'll use SQLAlchemy because it offers us a number of features and is reasonably popular. As with many things, there's no *best*; other ORM layers have different advantages and disadvantages.

Because of the popularity of using a relational database to support web development, web frameworks often include ORM layers. Django has its own ORM layer, as does `web.py`. In some cases, we can tease the ORMs out of the larger framework. However, it seems simpler to work with a standalone ORM.

The documentation, installation guide, and code for SQLAlchemy is available at `http://www.sqlalchemy.org`. When installing, using `--without-cextensions` can simplify the process if the high-performance optimizations aren't required.

It's important to note that SQLAlchemy can completely replace all of an application's SQL statements with first-class Python constructs. This has the profound advantage of allowing us to write applications in a single language, Python, even though a second language, SQL, is used under the hood as part of the data access layer. This can save some complexity in the development and debugging stages.

This does not, however, remove the obligation to understand the underlying SQL database constraints and how our design must fit within these constraints. An ORM layer doesn't magically obviate the design considerations. It merely changes the implementation language from SQL to Python.

We'll design an ORM-friendly class in the next section.

Designing ORM-friendly classes

When using an ORM, we will fundamentally change the way we design and implement our persistent classes. We're going to expand the semantics of our class definitions to have the following three distinct levels of meaning:

- The class will be used to create Python objects. The method functions are used by these objects.
- The class will also describe an SQL table and can be used by the ORM to create the SQL DDL that builds and maintains the database structure. The attributes will be mapped to SQL columns.
- The class will also define the mappings between the SQL table and Python class. It will be the vehicle to turn Python operations into SQL DML and build Python objects from SQL query results.

Most ORMs are designed to use descriptors to formally define the attributes of our class. We do not simply define attributes in the `__init__()` method. For more information on descriptors, see Chapter 4, *Attribute Access, Properties, and Descriptors*.

SQLAlchemy requires us to build a **declarative base class**. This base class provides a metaclass for our application's class definitions. It also serves as a repository for the metadata that we're defining for our database. If we follow the defaults, it's easy to call this class `Base`.

Here's the list of imports that might be helpful:

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Table
from sqlalchemy import (
    BigInteger,
    Boolean,
    Date,
    DateTime,
    Enum,
    Float,
    Integer,
    Interval,
    LargeBinary,
    Numeric,
    PickleType,
    SmallInteger,
    String,
```

```
    Text,
    Time,
    Unicode,
    UnicodeText,
    ForeignKey,
)
from sqlalchemy.orm import relationship, backref
```

We imported the essential definitions to create a column of a table, column, and to create the rare table that doesn't specifically map to a Python class, `Table`. We imported *all* of the generic column type definitions. We'll only use a few of these column types. Not only does SQLAlchemy define these generic types; it defines the SQL standard types, and it also defines vendor-specific types for the various supported SQL dialects. It is best to stick to the generic types and allow SQLAlchemy to map between generic, standard, and vendor types.

We also imported two helpers to define the relationships among tables, `relationship`, and `backref`. SQLAlchemy's metaclass is built by the `declarative_base()` function as follows:

```
Base = declarative_base()
```

The `Base` object must be the superclass for any persistent class that we're going to define. We'll define three tables that are mapped to Python classes. We'll also define a fourth table that's simply required by SQL to implement a many-to-many relationship.

Here's the `Blog` class:

```
class Blog(Base):
    __tablename__ = "BLOG"
    id = Column(Integer, primary_key=True)
    title = Column(String)

    def as_dict(self):
        return dict(
            title=self.title,
            underline="=" * len(self.title),
            entries=[e.as_dict() for e in self.entries],
        )
```

Our `Blog` class is mapped to a table named `"BLOG"`. We've included two descriptors for the two columns we want in this table. The `id` column is defined as an `Integer` primary key. Implicitly, this will be an autoincrement field so that surrogate keys are generated for us.

The title column is defined as a generic string. We could have used `Text`, `Unicode`, or even `UnicodeText` for this. The underlying engine might have different implementations for these various types. In our case, SQLite will treat all of these nearly identically. Also note that SQLite doesn't need an upper limit on the length of a column; other database engines might require an upper limit on the size of `String`.

The `as_dict()` method function refers to an `entries` collection that is clearly *not* defined in this class. When we look at the definition of the `Post` class, we'll see how this `entries` attribute is built. Here's the definition of the `Post` class:

```
class Post(Base):
    __tablename__ = "POST"
    id = Column(Integer, primary_key=True)
    title = Column(String)
    date = Column(DateTime)
    rst_text = Column(UnicodeText)
    blog_id = Column(Integer, ForeignKey("BLOG.id"))
    blog = relationship("Blog", backref="entries")
    tags = relationship("Tag", secondary=assoc_post_tag, backref="posts")

    def as_dict(self):
        return dict(
            title=self.title,
            underline="-" * len(self.title),
            date=self.date,
            rst_text=self.rst_text,
            tags=[t.phrase for t in self.tags],
        )
```

This class has five attributes, two relationships, and a method function. The `id` attribute is an integer primary key; this will have an autoincremented value by default. The `title` attribute is a simple string.

The `date` attribute will be a `DateTime` column; `rst_text` is defined as `UnicodeText` to emphasize our expectation of any Unicode character in this field.

The `blog_id` is a foreign key reference to the parent blog that contains this post. In addition to the foreign key column definition, we also included an explicit `relationship` definition between the post and the parent blog. This `relationship` definition becomes an attribute that we can use for navigation from the post to the parent blog.

The `backref` option includes a backward reference that will be added to the `Blog` class. This reference in the `Blog` class will be the collection of `Posts` that are contained within the `Blog`. The `backref` option names the new attribute in the `Blog`

class to reference the child `Posts`.

The `tags` attribute uses a `relationship` definition; this attribute will navigate via an association table to locate all the `Tag` instances associated with the post. We'll look at the following association table. This, too, uses `backref` to include an attribute in the `Tag` class that references the related collection of the `Post` instances.

The `as_dict()` method makes use of the `tags` attribute to locate all of `Tags` associated with this `Post`. Here's a definition for the `Tag` class:

```
class Tag(Base):
    __tablename__ = "TAG"
    id = Column(Integer, primary_key=True)
    phrase = Column(String, unique=True)
```

We defined a primary key and a `String` attribute. We included a constraint to ensure that each tag is explicitly unique. An attempt to insert a duplicate will lead to a database exception. The relationship in the `Post` class definition means that additional attributes will be created in this class.

As required by SQL, we need an association table for the many-to-many relationship between tags and posts. This table is purely a technical requirement in SQL and need not be mapped to a Python class:

```
assoc_post_tag = Table(
    "ASSOC_POST_TAG",
    Base.metadata,
    Column("POST_ID", Integer, ForeignKey("POST.id")),
    Column("TAG_ID", Integer, ForeignKey("TAG.id")),
)
```

We have to explicitly bind this to the `Base.metadata` collection. This binding is automatically a part of the classes that use `Base` as the metaclass. We defined a table that contains two `Column` instances. Each column is a foreign key to one of the other tables in our model.

Let's see how to build the schema with the ORM layer.

Building the schema with the ORM layer

In order to connect to a database, we'll need to create an engine. One use for the engine is to build the database instance with our table declarations. The other use for the engine is to manage the data from a session, which we'll look at later. Here's a script that we can use to build a database:

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///./p2_c11_blog2.db', echo=True)
Base.metadata.create_all(engine)
```

When we create an `Engine` instance, we use a URL-like string that names the vendor product and provides all the additional parameters required to create the connection to that database. In the case of SQLite, the connection is a filename. In the case of other database products, there might be server host names and authentication credentials.

Once we have the engine, we've done some fundamental metadata operations. We've shown you `create_all()`, which builds all of the tables. We might also perform a `drop_all()`, which will drop all of the tables, losing all the data. We can, of course, create or drop an individual schema item too.

If we change a table definition during software development, it will not automatically mutate the SQL table definition. We need to explicitly drop and rebuild the table. In some cases, we might want to preserve some operational data, leading to potentially complex surgery to create and populate new table(s) from old table(s).

The `echo=True` option writes log entries with the generated SQL statements. This can be helpful to determine whether the declarations are complete and create the expected database design. Here's a snippet of the output that is produced:

```
CREATE TABLE "BLOG" (
    id INTEGER NOT NULL,
    title VARCHAR,
    PRIMARY KEY (id)
)
```

```
CREATE TABLE "TAG" (
    id INTEGER NOT NULL,
    phrase VARCHAR,
    PRIMARY KEY (id),
    UNIQUE (phrase)
)

CREATE TABLE "POST" (
    id INTEGER NOT NULL,
    title VARCHAR,
    date DATETIME,
    rst_text TEXT,
    blog_id INTEGER,
    PRIMARY KEY (id),
    FOREIGN KEY(blog_id) REFERENCES "BLOG" (id)
)

CREATE TABLE "ASSOC_POST_TAG" (
    "POST_ID" INTEGER,
    "TAG_ID" INTEGER,
    FOREIGN KEY("POST_ID") REFERENCES "POST" (id),
    FOREIGN KEY("TAG_ID") REFERENCES "TAG" (id)
)
```

This shows SQL that the CREATE TABLE statements were created based on our class definitions. This can be helpful to see how the ORM definitions are implemented in a database.

Once the database has been built, we can create, retrieve, update, and delete objects. In order to work with database objects, we need to create a session that acts as a cache for the ORM-managed objects.

We'll see how to manipulate objects with the ORM layer in the next section.

Manipulating objects with the ORM layer

In order to work with objects, we'll need a session cache. This is bound to an engine. We'll add new objects to the session cache. We'll also use the session cache to query objects in the database. This assures us that all objects that need to be persistent are in the cache. Here is a way to create a working session:

```
from sqlalchemy.orm import sessionmaker
Session = sessionmaker(bind=engine)
session = Session()
```

We used the SQLAlchemy `sessionmaker()` function to create a `Session` class. This is bound to the database engine that we created previously. We then used the `Session` class to build a `session` object that we can use to perform data manipulation. A session is required to work with the objects in general.

Generally, we build one `sessionmaker` class along with the engine. We can then use that one `sessionmaker` class to build multiple sessions for our application processing.

For simple objects, we create them and load them into the session, as in the following code:

```
blog = Blog(title="Travel 2013")
session.add(blog)
```

This puts a new `Blog` object into the session named `session`. The `Blog` object is not *necessarily* written to the database. We need to commit the session before the database writes are performed. In order to meet the atomicity requirements, we'll finish building a post before committing the session.

First, we'll look up the `Tag` instances in the database. If they don't exist, we'll create them. If they do exist, we'll use the tag found in the database as follows:

```
tags = []
for phrase in "#RedRanger", "#Whitby42", "#ICW":
    try:
        tag = session.query(Tag).filter(Tag.phrase == phrase).one()
```

```
    except sqlalchemy.orm.exc.NoResultFound:
        tag = Tag(phrase=phrase)
        session.add(tag)
    tags.append(tag)
```

We use the `session.query()` function to examine instances of the given class. Each `filter()` function appends a criterion to the query. The `one()` function ensures that we've found a single row. If an exception is raised, then it means that `Tag` doesn't exist. We need to build a new `Tag` and add it to the session.

Once we've found or created the `Tag` instance, we can append it to a local list named `tags`; we'll use this list of `Tag` instances to create the `Post` object. Here's how we build a `Post`:

```
p2 = Post(date=datetime.datetime(2013,11,14,17,25),
    title="Hard Aground",
    rst_text="""Some embarrassing revelation. Including ☹ and ⚓""",
    blog=blog,
    tags=tags
    )
session.add(p2)
blog.posts = [p2]
```

This includes a reference to the parent blog. It also includes the list of `Tag` instances that we built (or found in the database).

The `Post.blog` attribute was defined as a relationship in the class definitions. When we assign an object, SQLAlchemy plucks out the proper ID values to create the foreign key reference that the SQL database uses to implement the relationship.

The `Post.tags` attribute was also defined as a relationship. The `Tag` objects are referenced via the association table. SQLAlchemy tracks the ID values properly to build the necessary rows in the SQL association table for us.

In order to associate the `Post` with the `Blog`, we'll make use of the `Blog.posts` attribute. This, too, was defined as a relationship. When we assign a list of `Post` objects to this relationship attribute, the ORM will build the proper foreign key reference in each `Post` object. This works because we provided the `backref` attribute when defining the relationship. Finally, we commit the session as follows:

```
session.commit()
```

The database inserts are all handled in a flurry of automatically generated SQL. The objects remained cached in the session. If our application continues using this session instance, then the pool of objects remains available without necessarily performing any actual queries against the database.

If, on the other hand, we would like to be absolutely sure that any updates written by other concurrent processes are included in a query, we can create a new, empty session for that query. When we discard a session and use an empty session, objects must be fetched from the database to refresh the session.

We can write a simple query as follows to examine and print all of the `Blog` objects:

```
session = Session()
for blog in session.query(Blog):
    print("{title}\n{underline}\n".format(**blog.as_dict()))
    for p in blog.entries:
        print(p.as_dict())
```

This will retrieve all the `Blog` instances. The `Blog.as_dict()` method will retrieve all of the posts within a blog. The `Post.as_dict()` will retrieve all of the tags. The SQL queries will be generated and executed automatically by SQLAlchemy.

We didn't include the rest of the template-based formatting from `Chapter 10`, *Serializing and Saving - JSON, YAML, Pickle, CSV, and XML*. It doesn't change. We are able to navigate from the `Blog` object via the `entries` list to the `Post` objects without writing elaborate SQL queries. Translating navigation into queries is the job of SQLAlchemy. Using a Python iterator is sufficient for SQLAlchemy to generate the right queries to refresh the cache and return the expected objects.

If we have `echo=True` defined for the `Engine` instance, then we'll be able to see the sequence of SQL queries performed to retrieve the `Blog`, `Post`, and `Tag` instances. This information can help us understand the workload that our application places on the database server process.

Let's see how to query posts that are given a tag.

Querying posts that are given a tag

An important benefit of a relational database is our ability to follow the relationships among the objects. Using SQLAlchemy's query capability, we can follow the relationship from `Tag` to `Post` and locate all `Posts` that share a given `Tag`.

A query is a feature of a session. This means that objects already in the session don't need to be fetched from the database, which is a potential time saver. Objects not in the session are cached in the session so that updates or deletes can be handled at the time of the commit.

To gather all of the posts that have a given tag, we need to use the intermediate association table as well as the `Post` and `Tag` tables. We'll use the query method of the session to specify what kinds of objects we expect to get back. We'll use the fluent interface to join in the various intermediate tables and the final table that we want with the selection criteria. Here's how it looks:

```
session2 = Session()
results = (
    session2.query(Post).join(assoc_post_tag).join(Tag).filter(
        Tag.phrase == "#Whitby42"
    )
)
for post in results:
    print(
        post.blog.title, post.date,
        post.title, [t.phrase for t in post.tags]
    )
```

The `session.query()` method specifies the table that we want to see. The `join()` methods identify the additional tables that must be matched. Because we provided the relationship information in the class definitions, SQLAlchemy can work out the SQL details required to use primary keys and foreign keys to match rows. The final `filter()` method provides the selection criteria for the desired subset of rows. Here's the SQL that was generated:

```
SELECT "POST".id AS "POST_id",
    "POST".title AS "POST_title",
    "POST".date AS "POST_date",
    "POST".rst_text AS "POST_rst_text",
    "POST".blog_id AS "POST_blog_id"
FROM "POST"
JOIN "ASSOC_POST_TAG" ON "POST".id = "ASSOC_POST_TAG"."POST_ID"
JOIN "TAG" ON "TAG".id = "ASSOC_POST_TAG"."TAG_ID"
```

```
WHERE "TAG".phrase = ?
```

The Python version is a bit easier to understand, as the details of the key matching can be elided. The `print()` function uses `post.blog.title` to navigate from the `Post` instance to the associated blog and show the `title` attribute. If the blog was in the session cache, this navigation is done quickly. If the blog was not in the session cache, it will be fetched from the database.

This navigation behavior applies to `[t.phrase for t in post.tags]` too. If the object is in the session cache, it's simply used. In this case, the collection of the `Tag` objects associated with a post might lead to a complex SQL query as follows:

```
SELECT
    "TAG".id AS "TAG_id",
    "TAG".phrase AS "TAG_phrase"
FROM "TAG", "ASSOC_POST_TAG"
WHERE ? = "ASSOC_POST_TAG"."POST_ID"
AND "TAG".id = "ASSOC_POST_TAG"."TAG_ID"
```

In Python, we simply navigated via `post.tags`. SQLAlchemy generated and executed the SQL for us.

Defining indices in the ORM layer

One of the ways to improve the performance of a relational database such as SQLite is to make join operations faster. The ideal way to do this is to include enough index information so that slow search operations aren't done to find matching rows.

When we define a column that might be used in a query, we should consider building an index for that column. This is a simple process that uses SQLAlchemy. We simply annotate the attribute of the class with `index=True`.

We can make fairly minor changes to our `Post` table, for example. We can do this to add indexes:

```
class Post(Base):
    __tablename__ = "POST"
    id = Column(Integer, primary_key=True)
    title = Column(String, index=True)
    date = Column(DateTime, index=True)
    blog_id = Column(Integer, ForeignKey('BLOG.id'), index=True)
```

Adding two indexes for the title and date will usually speed up queries for the posts by the title or by the date. There's no guarantee that there will be an improvement in the performance. Relational database performance involves a number of factors. It's important to measure the performance of a realistic workload both with the index and without it.

Adding an index by `blog_id`, similarly, might speed up the join operation between rows in the `Blog` and `Post` tables. It's also possible that the database engine uses an algorithm that doesn't benefit from having this index available.

Indexes involve storage and computational overheads. An index that's rarely used might be so costly to create and maintain that it becomes a problem, not a solution. On the other hand, some indexes are so important that they can have spectacular performance improvements. In all cases, we don't have direct control over the database algorithms being used; the best we can do is create the index and measure the performance impact.

Let's take a look at schema evolution in the next section.

Schema evolution

When working with an SQL database, we have to address the problem of schema evolution. Our objects have a dynamic state and a (relatively) static class definition. We can easily persist the dynamic state of an object. Our class defines the schema for the persistent data. The ORM provides mappings from our class to an SQL implementation.

When we change a class definition, how can we fetch objects from the database? If the database must change, how do we upgrade the Python mappings and still access the data? A good design often involves some combination of several techniques.

The changes to the methods and properties of a Python class will not change the mapping to the SQL rows. These can be termed minor changes, as the tables in the database are still compatible with the changed class definition. A new software release can have a new minor version number.

Some changes to Python class attributes will not necessarily change the persisted object state. Adding an index, for example, doesn't change the underlying table. SQL can be somewhat flexible when converting the data types from the database to the Python objects. An ORM layer can also add flexibility. In some cases, we can make some class or database changes and call it a minor version update because the existing SQL schema will still work with new class definitions.

Other changes to the SQL table definitions will need to involve modifying the persisted objects. These can be called major changes when the existing database rows will no longer be compatible with the new class definition. These kinds of changes should not be made by *modifying* the original Python class definitions. These kinds of changes should be made by defining a *new* subclass and providing an updated factory function to create instances of either the old or new class.

Tools such as Alembic (see `https://pypi.org/project/alembic/`) and Migrate (`https://sqlalchemy-migrate.readthedocs.io/en/latest/`) can help manage schema evolution. A disciplined history of schema migration steps is often essential for properly

converting from old data to new data.

There are two kinds of techniques used for transforming the schema from one version to the next. Any given application will use some combination of these techniques, as follows:

- SQL ALTER statements modify a table in place. There are a number of constraints and restrictions on what changes can be done with an ALTER. This generally covers a number of minor changes.
- Creating new tables and dropping old tables. Often, SQL schema changes require a new version of tables from data in the old tables. For a large database, this can be a time-consuming operation. For some kinds of structural changes, it's unavoidable.

SQL database schema changes typically involve running a one-time conversion script. This script will use the old schema to query the existing data, transform it to new data, and use the new schema to insert new data into the database. Of course, this must be tested on a backup database before being run on the user's live operational database. Once the schema change has been accomplished, the old schema can be safely ignored and later dropped to free up storage.

Tools such as Alembic require two separate conversion scripts. The upgrade script will move the schema forward to the new state. A downgrade script will move the schema back to the previous state. Debugging a migration may involve doing an upgrade, finding a problem, doing a downgrade, and using the previous software until the problem can be sorted out.

Summary

We looked at the basics of using SQLite in three ways: directly, via an access layer, and via the SQLAlchemy ORM. We have to create SQL DDL statements; we can do this directly in our applications or in an access layer. We can also have DDL built by the SQLAlchemy class definitions. To manipulate data, we'll use SQL DML statements; we can do this directly in a procedural style, or we can use our own access layer or SQLAlchemy to create the SQL.

Design considerations and tradeoffs

One of the strengths of the `sqlite3` module is that it allows us to persist distinct objects, each with its own unique change history. When using a database that supports concurrent writes, we can have multiple processes updating the data, relying on SQLite to handle concurrency via its own internal locking.

Using a relational database imposes numerous restrictions. We must consider how to map our objects to rows of tables in the database as follows:

- We can use SQL directly, using only the supported SQL column types and largely eschewing object-oriented classes.
- We can use a manual mapping that extends SQLite to handle our objects as SQLite BLOB columns.
- We can write our own access layer to adapt and convert between our objects and SQL rows.
- We can use an ORM layer to implement a row-to-object mapping.

Mapping alternatives

The problem with mixing Python and SQL is that there can be an impetus toward something that we might call the *All Singing, All Dancing, All SQL* solution. The idea here is that the relational database is somehow the ideal platform and Python corrupts this by injecting needless object-oriented features.

The all-SQL, object-free design strategy is sometimes justified as being more appropriate for certain kinds of problems. Specifically, the proponents will point out summarizing large sets of data using the SQL `GROUP BY` clause as an ideal use for SQL.

This kind of processing is implemented very effectively by Python's `defaultdict` and `Counter`. The Python version is often so effective that a small Python program querying lots of rows and accumulating summaries using `defaultdict` might be **faster** than a database server performing SQL with `GROUP BY`.

When in doubt, measure. When confronted with claims that SQL should magically be faster than Python, you should gather evidence. This data gathering is not confined to one-time initial technical spike situations, either. As usage grows and changes, the relative merit of SQL database versus Python will shift too. A home-brewed access layer will tend to be highly specific to a problem domain. This might have the advantage of high performance and relatively transparent mapping from row to object. It might have the disadvantage of being annoying to maintain every time a class changes or the database implementation changes.

A well-established ORM project might involve some initial effort to learn the features of the ORM, but the long-term simplifications are important benefits. Learning the features of an ORM layer can involve both initial work and rework as lessons are learned. The first attempts at a design that has good object features and still fits within the SQL framework will have to be redone as the application trade-offs and considerations become clearer.

Key and key design

Because SQL depends on keys, we must take care to design and manage keys for our various objects. We must design a mapping from an object to the key that will be used to identify that object. One choice is to locate an attribute (or combination of attributes) that contains proper primary keys and cannot be changed. Another choice is to generate surrogate keys that cannot be changed; this allows all other attributes to be changed.

Most relational databases can generate surrogate keys for us. This is usually the best approach. For other unique attributes or candidate key attributes, we can define SQL indexes to improve the processing performance.

We must also consider the foreign key relationships among objects. There are several common design patterns: one-to-many, many-to-one, many-to-many, and optional one-to-one. We need to be aware of how SQL uses keys to implement these relationships and how SQL queries will be used to fill in the Python collections.

Application software layers

Because of the relative sophistication available when using `sqlite3`, our application software must become more properly layered. Generally, we'll look at software architectures with the following layers:

- **The presentation layer:** This is a top-level user interface, either a web presentation or a desktop GUI.
- **The application layer:** This is the internal service or controllers that make the application work. This could be called the processing model, and is different from the logical data model.
- **The business layer or the problem domain model layer:** These are the objects that define the business domain or the problem space. This is sometimes called the logical data model. We looked at how we might model these objects using a microblog blog and post example.
- **Infrastructure:** This often includes several layers, as well as other cross-cutting concerns, such as logging, security, and network access.
- **The data access layer:** These are protocols or methods to access the data objects. It is often an ORM layer. We've looked at SQLAlchemy. There are numerous other choices for this.
- **The persistence layer:** This is the physical data model as seen in file storage. The `sqlite3` module implements persistence. When using an ORM layer such as SQLAlchemy, we only reference SQLite when creating an engine.

When looking at `sqlite3` in this chapter and `shelve` in `Chapter 11`, *Storing and Retrieving Objects via Shelve*, it becomes clear that mastering object-oriented programming involves some higher-level design patterns. We can't simply design classes in isolation—we need to look at how classes are going to be organized into larger structures.

Looking forward

In the next chapter, we'll look at transmitting and sharing objects using REST. This design pattern shows us how to manage the representation of the state and how to transfer the object state from process to process. We'll leverage a number of persistence modules to represent the state of an object that is being transmitted.

In `Chapter 14`, *Configuration Files and Persistence*, we'll look at the configuration files. We'll look at several ways to make use of persistent representations of data that controls an application.

Transmitting and Sharing Objects

We'll expand on our serialization techniques for the object representation started in `Chapter 10`, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML*. When we need to transmit an object, we perform some kind of **REpresentational State Transfer** (**REST**), which includes serializing a representation of the state of the object. This representation can then be transferred to another process (usually on another host computer); the receiving process can then build a version of the original object from the representation of the state and a local copy of the class definition.

Decomposing REST processing into the two aspects of representation and transfer lets us solve these problems independently. There are a variety of solutions that will lead to many workable combinations. We'll limit ourselves to two popular mechanisms: the RESTful web service, and the multiprocessing queue. Both will serialize and transmit objects between processes.

For web-based transfers, we'll leverage the **Hypertext Transfer Protocol** (**HTTP**). This allows us to implement the **Create-Retrieve-Update-Delete** (**CRUD**) processing operations based on the HTTP methods of `POST`, `GET`, `PATCH`, `PUT`, and `DELETE`. We can use this to build a RESTful web service. Python's **Web Service Gateway Interface** (**WSGI**) standard defines a general pattern for web services. Any practical application will use one of the available web frameworks that implement the WSGI standard. RESTful web services often use a JSON representation of object state.

In addition to HTTP, we'll look at local transfers among processes on the same host. This can be done more efficiently using local message queues provided by the `multiprocessing` module. There are numerous sophisticated third-party queue management products. We'll focus on the Python Standard Library offering.

There is an additional consideration when working with RESTful transfers: a client providing data to a server might not be trustworthy. To cope with this, we must implement some security in cases where untrustworthy data might be present. For some representations, specifically JSON, there are few security

considerations. YAML introduces a security concern and supports a safe load operation; see `Chapter 10`, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML*, for more information on this. Because of this security issue, the `pickle` module also offers a restricted unpickler that can be trusted to not import unusual modules and execute damaging code.

In this chapter, we will cover the following topics:

- Class, state, and representation
- Using HTTP and REST to transmit objects
- Using Flask to build a RESTful web service
- Handling stateful RESTful services
- Creating a secure RESTful service
- Implementing REST with a web application framework
- Using a message queue to transmit objects

Technical requirements

The code files for this chapter are available at `https://git.io/fj2U6`.

Class, state, and representation

Many applications can be decomposed into the processing done by servers and by clients. A single server will exchange data with multiple remote clients. In a hybrid situation, an application can be both a client of other remote computers, and a server to remote clients. There is an intentional asymmetry that is used to simplify the definition of the protocol: a client makes a request to the server, and the server responds to the client. The client initiates each request and waits for the response.

Separating clients and servers means that objects must be transmitted between the two processes. We can decompose the larger problem into two smaller problems. The inter-networking protocols define a way to transmit bytes from a process on one host to a process on another host. Serialization techniques transform our objects into bytes and then reconstruct the bytes from the objects. It helps, when designing classes, to focus on object **state** as the content exchanged between processes.

Unlike the object state, we transmit class definitions through an entirely separate method. Class definitions change relatively slowly, so we exchange the class definitions by the definition of the class in the form of the Python source. If we need to supply a class definition to a remote host, we can install the Python source code on that host.

When a client is written in a language other than Python, then an equivalent class definition must be provided. A JavaScript client, for example, will construct an object from the serialized JSON state of the Python object on the server. Two objects will have a **similar** state by sharing a common representation.

We're making a firm distinction between the entire, working object in Python's working memory, and the representation of the object's state that is transmitted. The whole Python object includes the class, superclasses, and other relationships in the Python runtime environment. The object's state may be represented by a simple string. Consider the following:

```
>>> from dataclasses import dataclass, asdict
>>> import json

>>> @dataclass
... class Greeting:
...     message: str

>>> g = Greeting("Hello World")
>>> text = json.dumps(asdict(g))
>>> text
'{"message": "Hello World"}'
>>> text.encode('utf-8')
b'{"message": "Hello World"}'
```

This example shows a simple class definition, `Greeting`, where the state is characterized by a single attribute value, `message`. The `asdict()` function, applied to an instance of a `dataclass`, creates a dictionary that we can serialize in JSON notation. Because the networks transmit bytes, the use of `text.encode()` creates a stream of bytes. This tiny example shows how the data class definition is entirely separate from the representation of the state of this example instance of the class.

Let's see how we can use HTTP and REST to transmit objects.

Using HTTP and REST to transmit objects

HTTP is defined through a series of **Request for Comments** (**RFC**) documents. We won't review all of the particulars, but we will touch on the three main points.

The HTTP protocol includes requests and replies. A request includes a method, a **Uniform Resource Identifier** (**URI**), some headers, and optional attachments. A number of available methods are defined in the standards. Most browsers focus on making `GET` and `POST` requests. The standard browsers include the `GET`, `POST`, `PUT`, and `DELETE` requests, which are the ones that we'll leverage, because they correspond to CRUD operations. We'll ignore most of the headers and focus on the path portion of the URI.

An HTTP reply includes a status code number and reason text, as well as any headers and attached data. There are a variety of status code numbers. The response codes are generally understood according to these patterns:

- The `1xx` codes are informational, and not used widely in RESTful services.
- The `2xx` replies indicate success.
- The `3xx` status codes indicate the redirection of a request to a different host or a different URI path.
- The `4xx` response codes tell the client that the request is erroneous, and the reply should include a more detailed error message.
- The `5xx` codes generally mean that the server has had some kind of problem.

Of these general ranges, we're interested in just a few, as follows:

- The `200` status code is the generic `OK` response from a server.
- The `201` status code is the `Created` response, which might be used to show us that a `POST` request worked and an object was successfully created.
- The `204` status code is the `No Content` response, which might be used for a `DELETE` request.
- The `400` status code is a `Bad Request` response, used to reject invalid data used

to `POST`, `PUT`, or `PATCH` an object.

- The `401` status code is `Unauthorized`; this would be used in a secure environment to reject invalid credentials. It may also be used if valid user credentials are used, but the user lacks the authorization to take the action they requested.
- The `404` status code is `Not Found`, which is generally used when the URI path information does not identify a resource.

HTTP is defined as stateless. The server is not expected to have any recollection of previous interactions with a client. In some cases, this is a profound limitation, and there are several commonly used workarounds to manage state by exchanging state details with the client of the transaction. For interactive websites, cookies are sent from the server to the client. The client embeds the cookies with the request to the server, allowing the server to recover the transaction state and offer rich, stateful application behavior.

Cookies are often used when a web server provides the user experience through HTML forms. By sending cookies back to the browser, a server can track user login and session information. As the user takes actions, the server records details in the session objects that are serialized as cookies.

For RESTful web services, however, the client will not be a person sitting at a browser. The client of a RESTful service will be an application that can maintain the state of the user experience. This means that RESTful services can leverage simpler, stateless HTTP without cookies. This also means that states such as **logged-in** and **logged-out** don't apply to web services. For authentication purposes, credentials of some kind are often provided with each request. This imposes an obligation to secure the connection. In practice, all RESTful web servers will use a **Secure Sockets Layer** (**SSL**) and an HTTPS connection.

In the next section, we'll look at how to implement CRUD operations via REST.

Implementing CRUD operations via REST

We'll look at three fundamental ideas behind the REST protocols. The first idea is to use the text serialization of an object's state. The second is to use the HTTP request URI to name an object; a URI can include any level of detail, including a schema, module, class, and object identity in a uniform format. Finally, we can use the HTTP method to map to CRUD rules to define the action to be performed on the named object.

The use of HTTP for RESTful services pushes the envelope of the original definitions of HTTP requests and replies. This means that some of the request and reply semantics are open to active, ongoing discussion. Rather than presenting all of the alternatives, we'll suggest an approach. Our focus is on the Python language, not the more general problem of designing RESTful web services. A REST server will often support CRUD operations via the following five abstract use cases:

- **Create**: We'll use an HTTP `POST` request to create a new object and a URI that provides class information only. A path such as `/app/blog/` might name the class. The response could be a `201` message that includes a copy of the object as it was finally saved. The returned object information may include the URI assigned by the RESTful server for the newly created object or the relevant keys to construct the URI. A `POST` request is expected to change the RESTful resources by creating something new.
- **Retrieve – Search**: This is a request that can retrieve multiple objects. We'll use an HTTP `GET` request and a URI that provides search criteria, usually in the form of a query string after the `?` character. The URI might be `/app/blog/?title="Travel 2012-2013"`. Note that `GET` never makes a change to the state of any RESTful resources.
- **Retrieve – Single Instance**: This is a request for a single object. We'll use an HTTP `GET` request and a URI that names a specific object in the URI path. The URI might be `/app/blog/id/`. While the response is expected to be a single object, it might still be wrapped in a list to make it compatible with a search response. As this response is `GET`, there's no change in the state.

- **Update**: We'll use an HTTP `PUT` request and a URI that identifies the object to be replaced. We can also use an HTTP `PATCH` request with a document payload that provides an incremental update to an object. The URI might be `/app/blog/id/`. The response could be a `200` message that includes a copy of the revised object.
- **Delete**: We'll use an `HTTP DELETE` request and a URI that looks like `/app/blog/id/`. The response could be a simple `204 NO CONTENT` message without any object details in the response.

As the HTTP protocol is stateless, there's no provision for logon and logoff. Each request must be separately authenticated. We will often make use of the HTTP `Authorization` header to provide the username and password credentials. When doing this, we absolutely must also use SSL to provide security for the content of the `Authorization` header. There are more sophisticated alternatives that leverage separate identity management servers to provide authentication tokens rather than credentials.

The next section shows how to implement non-CRUD operations.

Implementing non-CRUD operations

Some applications will have operations that can't easily be characterized via CRUD verbs. We might, for example, have a **Remote Procedure Call** (**RPC**) style application that performs a complex calculation. Nothing is really created on the server. We might think of RPC as an elaborate retrieve operation where the calculation's arguments are provided in each request.

Most of the time, these calculation-focused operations can be implemented as the `GET` requests, where there is no change to the state of the objects in the server. An RPC-style request is often implemented via the HTTP `POST` method. The response can include a **Universally Unique Identifier** (**UUID**) as part of tracking the request and responses. It allows for the caching of responses in the cases where they are very complex or take a very long time to compute. HTTP headers, including `ETag` and `If-None-Match`, can be used to interact with the caching to optimize performance.

This is justified by the idea of preserving a log of the request and reply as part of a non-repudiation scheme. This is particularly important in websites where is a fee is charged for the services.

Let's take a look at the REST protocol and ACID in the next section.

The REST protocol and ACID

The ACID properties were defined in `Chapter 11`, *Storing and Retrieving Objects via Shelve*. These properties can be summarized as **Atomic**, **Consistent**, **Isolated**, and **Durable**. These are the essential features of a transaction that consists of multiple database operations. These properties don't automatically become part of the REST protocol. We must consider how HTTP works when we also need to ensure that the ACID properties are met.

Each HTTP request is atomic; therefore, we should avoid designing an application that makes a sequence of related `POST` requests and hopes that the individual steps are processed as a single, atomic update. Instead, we should look for a way to bundle all of the information into a single request to achieve a simpler, atomic transaction.

Additionally, we have to be aware that requests will often be interleaved from a variety of clients; therefore, we don't have a tidy way to handle isolation among interleaved sequences of requests. If we have a properly multilayered design, we should delegate the durability to a separate persistence module. In order to achieve the ACID properties, a common technique is to define bodies for the `POST`, `PUT`, or `DELETE` requests that contain *all* the relevant information. By providing a single composite object, the application can perform all of the operations in an atomic request. These larger objects become documents that might contain several items that are part of a complex transaction.

When looking at our blog and post relationships, we see that we might want to handle two kinds of HTTP `POST` requests to create a new `Blog` instance. The two requests are as follows:

- **A blog with only a title and no additional post entries**: We can easily implement the ACID properties for this, as it's only a single object.
- **A composite object that is a blog plus a collection of post entries**: We need to serialize the blog and all of the relevant `Post` instances. This needs to be sent as a single `POST` request. We can then implement the ACID properties by creating the blog, the related posts, and returning a single `201 Created` status when the entire collection of objects have been made durable. This

may involve a complex multi-statement transaction in the database that supports the RESTful web server.

Let's look at how we can choose a representation out of JSON, XML, or YAML.

Choosing a representation – JSON, XML, or YAML

There's no clear reason to pick a single representation; it's relatively easy to support a number of representations. The client should be permitted to demand a representation. There are several places where a client can specify the representation:

- The client can use a part of a query string, such as `https://host/app/class/id/?form=XML`. The portion of the URL after `?` uses the `form` value to define the output format.
- The client can use a part of the URI, such as `https://host/app;XML/class/id/`. The `app;XML` syntax names the application, `app`, and the format, `XML`, by using a sub-delimiter of `;` within the path.
- The client can use the `https://host/app/class/id/#XML` fragment identifier. The portion of the URL after `#` specifies a fragment, often a tag on a heading within an HTML page. For a RESTful request, the `#XML` fragment provides the format.
- The client can use a separate header. The `Accept` header, for example, can be used to specify the representation as part of the `MIME` type. We might include `Accept: application/json` to specify that a JSON-formatted response is expected.

None of these are obviously superior, but compatibility with some existing RESTful web services may suggest a particular format. The relative ease with which a framework parses a URI pattern may also suggest a particular format.

JSON is preferred by many JavaScript presentation layers. Other representations, such as XML or YAML, can be helpful for other presentation layers or other kinds of clients. In some cases, there may be yet another representation. For example, MXML or XAML might be required by a particular client application.

In the next section, we'll see how to use Flask to build a RESTful web service.

Using Flask to build a RESTful web service

Since the REST concepts are built on the HTTP protocol, a RESTful API is an extension to an HTTP service. For robust, high-performance, secure operations, a common practice is to build on a server such as **Apache HTTPD** or **NGINX**. These servers don't support Python directly; they require an extension module to interface with a Python application.

Any interface between externally-facing web servers and Python will adhere to the **Web Services Gateway Interface** (**WSGI**). For more information, see `http://www.wsgi.org`. The Python standard library includes a WSGI reference implementation. See PEP 3333, `http://www.python.org/dev/peps/pep-3333/`, for details on this standard. See `https://wiki.python.org/moin/WebServers` for a number of web servers that support WSGI. When working with NGINX, for example, the **uWSGI** plugin provides the necessary bridge to Python.

The WSGI standard defines a minimal set of features shared by all Python web frameworks. It's challenging to work with, however, because many features of web services aren't part of this minimal interface. For example, authentication and session management can be implemented in a way that adheres to the WSGI standard, but it can be rather complex.

There are several high-level application frameworks that adhere to the WSGI standard. When we build a RESTful web service, we'll often use a framework that makes it very easy to create our application. For the examples in this chapter, we'll use the Flask framework. For more information, see `http://flask.pocoo.org/docs/1.0/`.

A Flask application is an instance of the `Flask` class. An essential feature of Flask is the routing table to map URI paths to specific functions. We'll start with a simple application with a few routes to show how this works. First, however, we need some objects to transfer via RESTful services.

Let's take a look at the problems faced while transferring domain objects.

Problem-domain objects to transfer

A RESTful web server works by transferring representations of object state. We'll define a few simple objects to transfer from a RESTful server to a RESTful client. This application serves a collection of tuples that represent dominoes. We'll include a property to discern whether a domino has the same value on both sides; a piece sometimes called a **spinner** or a **double**. The core definition of the `Domino` class is as follows:

```
from dataclasses import dataclass, asdict, astuple
from typing import List, Dict, Any, Tuple, NamedTuple
import random

@dataclass(frozen=True)
class Domino:
    v_0: int
    v_1: int

    @property
    def double(self):
        return self.v_0 == self.v_1

    def __repr__(self):
        if self.double:
            return f"Double({self.v_0})"
        else:
            return f"Domino({self.v_0}, {self.v_1})"
```

When we play a game of dominoes, the tiles are often shuffled by the players and dealt into hands. The remaining dominoes form a pile sometimes called a **boneyard**. This is analogous to a deck of cards, but dominoes aren't often stacked, and can instead be left lying in a corner of the table. In some four-player games, all 28 dominoes are used. The `Boneyard` class definition is as follows:

```
class Boneyard:

    def __init__(self, limit=6):
        self._dominoes = [
            Domino(x, y) for x in range(0, limit + 1) for y in range(0, x + 1)
        ]
        random.shuffle(self._dominoes)

    def deal(self, tiles: int = 7, hands: int = 4) -> List[List[Tuple[int, int]]]:
        if tiles * hands > len(self._dominoes):
            raise ValueError(f"tiles={tiles}, hands={hands}")
        return [self._dominoes[h:h + tiles]
            for h in range(0, tiles * hands, tiles)]
```

The individual `Domino` instances are created from a rather complex-looking list comprehension. The two `for` clauses can be transformed into `for` statements that look like the following:

```
for x in range(0, limit+1):
    for y in range(0, x+1):
        Domino(x, y)
```

This design assures that $x \geq y$; it will generate a `Domino(2, 1)` instance but will not create a `Domino(1, 2)` instance. This will create the 28 dominoes in a standard **double-six** set.

Once the dominoes are shuffled, then dealing them is a matter of taking slices from the list. When dealing hands of seven tiles, the `h` variable has values of 0, 7, 14, and 21. This leads to slices as `[0: 7]`, `[7: 14]`, `[14: 21]`, and `[21: 28]`; this divides the pool into four hands of seven tiles each.

A simple demonstration of how this works is shown as follows:

```
>>> random.seed(2)
>>> b = Boneyard(limit=6)
>>> b.deal(tiles=7, hands=2)
[[Domino(2, 0), Double(5), Domino(5, 2), Domino(5, 0), Double(0), Domino(6, 3), Domino(2
```

First, a `Boneyard` object is built; the limit value defines this as a double-six set of 28 individual pieces. When two hands of seven tiles are dealt, a predictable pair of hands are created because the random number generator is seeded with a fixed value, two, for the seed.

In some games, the highest double determines who plays first. In these two hands, the player with `Double(5)` would be the first to play.

Let's look at how we can create a simple application and server.

Creating a simple application and server

We'll write a very simple REST server that provides a series of hands of dominoes. This will work by routing a URI to a function that will provide the hands. The function must create a response object that includes one or more of the following items:

- A status code: The default is `200`, which indicates success.
- Headers: The default is a minimal set of response headers with the content size.
- Content: This can be a stream of bytes. In many cases, RESTful web services will return a document using JSON notation. The Flask framework provides us with a handy function for converting objects to JSON Notation, `jsonify()`.

A Flask application that deals a simple hand of dominoes can be defined as follows:

```
from flask import Flask, jsonify, abort
from http import HTTPStatus

app = Flask(__name__)

@app.route("/dominoes/<n>")
def dominoes(n: str) -> Tuple[Dict[str, Any], int]:
    try:
        hand_size = int(n)
    except ValueError:
        abort(HTTPStatus.BAD_REQUEST)

    if app.env == "development":
        random.seed(2)
    b = Boneyard(limit=6)
    hand_0 = b.deal(hand_size)[0]
    app.logger.info("Send %r", hand_0)

    return jsonify(status="OK", dominoes=[astuple(d) for d in hand_0]), HTTPStatus.OK
```

This shows us some of the ingredients in a Flask application. The most essential ingredient is the `Flask` object itself, assigned to an `app` variable. In small applications, this is often located in a module that provides a useful `__name__` string for the application. In larger applications, a `__name__` string may be

more helpful than a module name to help identify log messages from the `Flask` object. If the application is assigned to the `app` variable, then the automated Flask execution environment will locate it. If we don't call it `app`, we'll have to provide a function to start the execution.

The `@app.route` decorator is used for each function that will handle requests and create responses. There are a number of features of the route definition. In this example, we've used the parsing capability. The second item on the path is separated and assigned to the `n` parameter for the `dominoes()` function.

Generally, there are four important steps in completing a RESTful transaction:

- **Parsing:** The routing did some of the parsing. After the initial path decomposition, the parameter value was checked to be sure that it would lead to valid behavior. In case of a problem, the `abort()` function is used to return an HTTP status code showing a bad request.
- **Evaluating:** This section computes the response. In this case, a fresh set of dominoes is created. The limit of double-six is hard-wired into this application. To make this more useful for other games, the limit should be a configuration value. The `deal()` method of the `Boneyard` class creates a list of hands. This function, however, only returns a single hand, so `hand_0` is assigned the first of the hands in the list returned by `deal()`.
- **Logging**. The Flask logger is used to write a message to a log showing the response. In larger applications, the logging will be more complex. In some cases, there can be multiple logs to provide separate details of authentication or audit history.
- **Responding**. The response from a function in Flask can have from one to three items in it. In this case, two values are provided. The `jsonify()` function is used to return the JSON representation of a dictionary with two keys: `"status"`, and `"dominoes"`. The status code is built from the value of `HTTPStatus.OK`. Note that each of the `Domino` objects were converted into a tuple using the `dataclasses.astuple()` function. These kind of serialization considerations are an important part of REST.

The type hints that Flask functions use are generally very simple. Most functions in a RESTful application will have some combination of the following results:

- `Dict[str, Any]`: This is the simple result produced by `jsonify()`. It will yield the default status of `HTTPStatus.OK`.

- `Tuple[Dict[str, Any], int]`: This is a result with a non-default status code.
- `Tuple[Dict[str, Any], int, Dict[str, str]]`: This is a result with headers in addition to a status code and a document.

Other combinations of return values are possible, but are relatively rare. A function that implements a deletion, for example, might return only `HTTPStatus.NO_CONTENT` to show success without any further details.

We can start a demonstration version of this server from the Bash or Terminal prompt as follows:

```
$ FLASK_APP=ch13_ex2.py FLASK_ENV=development python -m flask run
```

This command sets two environment variables, then runs the `flask` module. The `FLASK_APP` variable defines which module contains the `app` object. The `FLASK_ENV` environment variable sets this as a `development` server, providing some additional debugging support. This will produce an output similar to the following:

```
 * Serving Flask app "ch13_ex2.py" (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 154-541-338
```

The `Flask app` name, `ch13_ex2.py`, came from the `FLASK_APP` environment variable. The `development` environment came from the `FLASK_ENV` environment variable. `debug mode` was enabled because the environment was `development`. We can browse the given URL to interact with those portions of the application that happen to be browser-friendly.

This application responds to `GET` requests. We can use a browser to perform a request on `http://127.0.0.1:5000/dominoes/7` and see a result similar to the following:

```
{"dominoes": [[2, 0], [5, 5], [5, 2], [5, 0], [0, 0], [6, 3], [2, 1]], "status": "OK"}
```

In the `development` mode, the random number generator is seeded with a fixed value, so the value is always as shown in the preceding code.

The response document shown above contains two features that are typical of RESTful web services:

- **A status item**: This is a summary of the response. In some cases, the HTTP status code will provide similar information. In many cases, this will be far more nuanced than the simplistic HTTP status code of `200` OK.
- **The requested object**: In this case, `dominoes` is a list of two-tuples. This is a representation of the state of a single hand. From this, the original `Domino` instances can be reconstructed by a client.

The original documents can be rebuilt with the following code:

```
document = response.get_json()
hand = list(Domino(*d) for d in document['dominoes'])
```

The client software must include the capability of parsing the JSON from the body of the response. The `Domino` class definition is used to reproduce objects from the transferred representation.

In the next section, we'll discuss more sophisticated routing and responses.

More sophisticated routing and responses

We'll look at two additional features common to Flask applications and RESTful APIs:

- The first is more sophisticated path parsing.
- The second is an Open API specification document.

Flask provides several clever parsing features to decompose the path in a URI. Perhaps one of the most useful parsers is used to extract individual field items from the URL string. We can use this as shown in the following example:

```
@app.route("/hands/<int:h>/dominoes/<int:c>")
def hands(h: int, c: int) -> Tuple[Dict[str, Any], int]:
    if h == 0 or c == 0:
        return jsonify(
            status="Bad Request", error=[f"hands={h!r}, dominoes={c!r} is invalid"]
        ), HTTPStatus.BAD_REQUEST

    if app.env == "development":
        random.seed(2)
    b = Boneyard(limit=6)
    try:
        hand_list = b.deal(c, h)
    except ValueError as ex:
        return jsonify(
            status="Bad Request", error=ex.args
        ), HTTPStatus.BAD_REQUEST
    app.logger.info("Send %r", hand_list)

    return jsonify(
        status="OK",
        dominoes=[[astuple(d) for d in hand] for hand in hand_list]
    ), HTTPStatus.OK
```

This uses `<int:h>` and `<int:c>` to both parse the values from the path and ensure that the values are integers. Any requests with paths that don't match this pattern will receive a `404 Not Found` error status.

The four steps of handling the response are slightly more complex. The details are as follows:

- The **parsing** step includes a range check on the parameter values. In the

case of a `0` value, this will lead to `400` Bad Request errors. The pattern of the request fits the route's template, but the values were invalid.

- The **evaluating** step includes an exception handler. If the underlying application model, the `Boneyard` object in this example, raises an exception, this Flask container will produce a useful, informative error message. Without this exception-handling block, `ValueError` would lead to an HTTP status code of `500` Internal Server Error.
- The **logging** step didn't change from the previous example.
- The **responding** step builds the list of individual hands by dealing tiles. This uses a more complex nested list comprehension. The nested comprehension can be read from right to left. The Python syntax is as follows:

```
[[astuple(d) for d in hand] for hand in hand_list]
```

  If we read the code in reverse, we can see that the nested comprehension has the following structure. It processes each `hand` in `hand_list`. Given `hand`, it then processes each `domino` in `hand`. The processing is provided first, which represents the `domino` object, `d`, as a tuple.

The goal of most RESTful web services development is to create an underlying application model that permits simple wrapper functions such as the following example.

An additional feature that is central to a successful RESTful server is a specification of the contract that the server adheres to. A document that follows the OpenAPI specification is a useful way to present this contract. For more information on the Open API specification, see https://swagger.io/specification/. The document service will look as follows:

```
OPENAPI_SPEC = {
    "openapi": "3.0.0",
    "info": {
        "description": "Deals simple hands of dominoes",
        "version": "2019.02",
        "title": "Chapter 13. Example 2",
    },
    "paths": {}
}

@app.route("/openapi.json")
def openapi() -> Dict[str, Any]:
    return jsonify(OPENAPI_SPEC)
```

The specification in this example is a tiny skeleton with only a few required fields. In many applications, the specification will be kept in a static file and

served separately. In very complex environments, there will be external references and the document may be provided in multiple, separate sections. To an extent, it's possible to deduce elements of the OpenAPI specification from the Flask routes themselves.

While this is technically possible, it subverts the idea of having a formal and independent contract for the features of the RESTful API. It seems better to have an external contract and check compliance against that contract.

The function that implements the route does almost nothing to transform the Python dictionary into JSON notation and respond with a document and the default status of `200` OK.

We'll see how to implement a REST client in the next section.

Implementing a REST client

Before looking at a smarter REST server application, we'll look at writing a REST client. The standard library has two packages useful for creating a client: `http.client` and `urllib`. These packages both provide features required to make RESTful requests over HTTP. Both packages include features to create requests with bodies, include headers, add complex URLs and retrieve a result, save the result headers, save any attachments, and track the status code. A popular alternative to `urllib` is the `requests` package. For further information, check out http://docs.python-requests.org/en/master/.

You can get the Open API specification with the following simple code:

```
import requests
get_openapi = requests.get(
    "http://127.0.0.1:5000/openapi.json")
if get_openapi.status_code == 200:
    document = get_openapi.json()
```

This will make an HTTP `GET` request to the given URI. The response object, `get_openapi`, will be examined and a few useful fields extracted. The value of `get_openapi.status_code` is the HTTP status code returned from the server. The `get_openapi.json()` method will attempt to read and parse the body of the response. If it is valid JSON, the resulting Python object can be examined further. In the case of Open API specifications, the object will be a Python dictionary. For these examples, we'd expect `document['info']['version'] != '2019.02'`. Any other result indicates a potential problem between the client and server.

In the next section, we'll unit test the RESTful services.

Demonstrating and unit testing the RESTful services

For proper unit testing, we want a more formal exchange between a client and a server. The Flask framework provides `test_client()`, which gives us the features required to make requests of a Flask server to be sure that the responses are correct. Once a test client object has been created, we can make our `get`, `post`, `put`, `patch`, and `delete` requests to confirm that a function behaves correctly.

The `doctest` module will locate Python examples inside document strings for a module or function. This makes it easy to include examples inside the definition of the relevant function. Here's an example of using the `doctest` features to test a Flask module:

```
@app.route("/openapi.json")
def openapi() -> Dict[str, Any]:
    """
    >>> client = app.test_client()
    >>> response = client.get("/openapi.json")
    >>> response.get_json()['openapi']
    '3.0.0'
    >>> response.get_json()['info']['title']
    'Chapter 13. Example 2'
    """
    return jsonify(OPENAPI_SPEC)
```

This test code uses `app.test_client()` to create a client. Then, `client.get()` is used to execute a `GET` request to the `/openapi.json` path. The results can be examined to be sure that an Open API specification document really will be provided by the defined route. This test case depends on the value of `app` being the Flask application module. This is usually true, because this code is in the same module where the `app` object is defined.

There are a variety of approaches to packaging unit tests:

- In a separate `tests` package: This will contain a number of test modules. If the filenames all begin with `test_`, then tools such as `pytest` can locate and execute these tests. This is appropriate when tests are relatively complex. This often happens when mock objects must be provided.

- In module and function docstrings: This is the technique that was shown previously. The `doctest` tool can locate examples within docstrings and execute these tests. The `doctest` module can also locate tests as the values of a global `__test__` dictionary. This is appropriate when the tests are relatively simple and the setup for each test is easily accomplished through application configuration.

The preceding test can be executed from the command line by using a command such as the following:

```
$ python3 -m doctest ch13_ex2.py
```

This command runs the `doctest` module. The command-line argument is the module to be examined. All Python examples marked with `>>>` will be executed to confirm that the function operates properly.

Handling stateful REST services

A RESTful server is often called upon to maintain the state of the resources it manages. The canonical CRUD operations map nicely to the HTTP methods available in RESTful processing. There may be `POST` requests to create new resources, `PATCH` requests to update those resources, and `DELETE` requests to remove resources.

In order to properly work with individual resources, unique resource identification becomes important. We'll look into RESTful object identification, before looking at the Flask server design that works with multiple objects.

Designing RESTful object identifiers

Object serialization involves defining some kind of identifier for each object. For `shelve` or `sqlite`, we need to define a string key for each object. A RESTful web server makes the same kinds of demands to define a workable key that can be used to unambiguously track down objects. A simple, surrogate key can also work out for a RESTful web service identifier. It can easily parallel the key used for `shelve` or `sqlite`.

What's important is this idea: *cool URIs don't change*. See http://www.w3.org/Provider/Style/URI.html.

It is important for us to define a URI that is never going to change. It's essential that the stateful aspects of an object are never used as part of the URI. For example, a microblogging application may support multiple authors. If we organize blog posts into folders by the author, we create problems for shared authorship and we create larger problems when one author takes over another author's content. We don't want the URI to switch when a purely administrative feature, such as the ownership, changes.

A RESTful application may offer a number of indices or search criteria. However, the essential identification of a resource or object should never change as the indices are changed or reorganized.

For relatively simple objects, we can often identify some sort of identifier – often a database surrogate key. In the case of blog posts, it's common to use a publication date (as that can't change) and a version of the title with punctuation and spaces replaced by _ characters. The idea is to create an identifier that will not change, no matter how the site gets reorganized. Adding or changing indexes can't change the essential identification of a microblog post.

For more complex objects that are containers, we have to decide on the granularity with which we will refer to these more complex objects. Continuing the microblog example, we have blogs as a whole, which contain a number of individual posts.

The URI for a blog can be something simple like this:

```
/microblog/blog/bid/
```

The top-most name (`microblog`) is the overall application. Then, we have the type of resource (`blog`) and finally, an ID, *`bid`*, for a specific instance.

The URI names for a post, however, have several choices:

```
/microblog/post/title_string/
/microblog/post/bid/title_string/
/microblog/blog/bid/post/title_string/
```

The first URI doesn't work well if multiple posts have the same title. In this case, something would have to be done to make the title unique. An author may see their title made unique with an extra `_2` or some other decoration. This is often undesirable.

The second URI uses the blog ID (`bid`) as a context or namespace to ensure that the `Post` titles are treated as unique within the context of a blog. This kind of technique is often extended to include additional subdivisions, such as a date, to further shrink the search space. This example is a little awkward, because the classification, `post`, is followed by the blog's ID.

The third example uses an explicit class/object naming at two levels: `blog/bid` and `post/title_string`. This has the disadvantage of a longer path, but it has the advantage of allowing a complex container to have multiple items in distinct internal collections.

Note that REST services have the effect of defining paths to resources in persistent storage. The URIs must be chosen with an eye toward clarity, meaning, and durability. A cool URI doesn't change.

Let's learn about the different layers of REST services.

Multiple layers of REST services

It's common for a complex RESTful application to use a database to save persistent objects, as well as change the state of those persistent objects. This often leads to a multi-tier web application. One view of a design will focus on these three tiers:

- Presentation to people is handled by HTML pages, possibly including JavaScript. The HTML content often starts as templates, and a tool such as Jinja (`http://jinja.pocoo.org`) is used to inject data into the HTML template. In many cases, the presentation layer is a separate Flask application focused on supporting features of a rich user experience, including stateful sessions and secure authentication. This layer will make a request to the application layer via RESTful requests.
- Application processing is done via a RESTful server that models the problem domain. Concerns such as user authentication or stateful interactions are not part of this layer.
- Persistence is done via a database of some kind. In many practical applications, a complex module will define an `init_app()` function to initialize a database connection.

As we saw in `Chapter 11`, *Storing and Retrieving Objects with Shelve*, it's quite easy to have a database that has an interface similar to a dictionary. We'll leverage this simple-looking interface to create a RESTful service that tracks a stateful object.

Using a Flask blueprint

Before defining the blueprint, we'll define some classes and functions that establish the problem domain. We'll define an enumeration of the `Status` values, with `"Updated"` and `"Created"` as the only two values. We'll define a set of dice using a dataclass named `Dice`. This class includes the current state of the dice, a unique identifier for this particular collection of dice, and an overall status to show whether it was initially created or has been updated. Separate from the class, it's helpful to have a `make_dice()` function to create a `Dice` instance.

For this example, here are the definitions:

```
from typing import Dict, Any, Tuple, List
from dataclasses import dataclass, asdict
import random
import secrets
from enum import Enum

class Status(str, Enum):
    UPDATED = "Updated"
    CREATED = "Created"

@dataclass
class Dice:
    roll: List[int]
    identifier: str
    status: str

    def reroll(self, keep_positions: List[int]) -> None:
        for i in range(len(self.roll)):
            if i not in keep_positions:
                self.roll[i] = random.randint(1, 6)
        self.status = Status.UPDATED

def make_dice(n_dice: int) -> Dice:
    return Dice(
        roll=[random.randint(1, 6) for _ in range(n_dice)],
        identifier=secrets.token_urlsafe(8),
        status=Status.CREATED
    )
```

The `Dice` class represents a handful of six-sided dice. A particular roll of the dice also has an identifier; this is a surrogate key used to identify an initial toss of the dice. Selected dice can be rerolled, which is consistent with a number of dice games.

The status attribute is used as part of a RESTful response to show the current

state of the object. The status attribute is either the `"Created"` or `"Updated"` string based on the `Status` class definition.

The `make_dice()` function creates a new `Dice` instance. It's defined outside the `Dice` class definition to emphasize its roll as creating a `Dice` instance. This could be defined as a method within the `Dice` class, and decorated with the `@classmethod` or `@staticmethod` decorator.

While a very strict object-oriented design approach mandates that everything be part of a class, Python imposes no such restriction. It seems simpler to have a function outside a class, rather than a decorated function within the class. Both are defined in a common module, so the relationship between function and class is clear.

The `reroll()` method of a `Dice` instance updates the object's internal state. This is the important new feature in this section.

The Open API Specification for this service has a skeleton definition as follows:

```
OPENAPI_SPEC = {
    "openapi": "3.0.0",
    "info": {
        "title": "Chapter 13. Example 3",
        "version": "2019.02",
        "description": "Rolls dice",
    },
    "paths": {
        "/rolls": {
            "post": {
                "description": "first roll",
                "responses": {201: {"description": "Created"}},
            },
            "get": {
                "description": "current state",
                "responses": {200: {"description": "Current state"}},
            },
            "patch": {
                "description": "subsequent roll",
                "responses": {200: {"description": "Updated"}},
            }
        }
    }
}
```

This specification provides one path, `/rolls`, which responds to a variety of method requests. A `post` request has a very terse description of `"first roll"`; only one of the possible responses is defined in this example specification. Similarly, the `get` and `patch` requests have the minimal definition required to pass a simple

schema check.

This specification omits the lengthy definition of the parameters required for the `post` and `patch` requests. The `post` request requires a document in JSON Notation. Here's an example: `{"dice": 5}`. The `patch` request also requires a document in JSON Notation, the body of this document specifies which dice must be left alone, and which must be re-rolled. The content of the document only specifies which dice to keep; all of the others will be re-rolled. It will look like this: `{"keep": [0, 1, 2]}`.

One of the tools for decomposing a complex Flask application is a blueprint. A blueprint is registered with a Flask application with a specific path prefix. This allows us to have multiple, related portions of a complex application by using several instances of a common blueprint.

For this example, we'll only register a single instance of a `Blueprint` object. The definition of a `Blueprint` object starts as follows:

```
from flask import Flask, jsonify, request, url_for, Blueprint, current_app, abort
from typing import Dict, Any, Tuple, List

SESSIONS: Dict[str, Dice] = {}

rolls = Blueprint("rolls", __name__)

@rolls.route("/openapi.json")
def openapi() -> Dict[str, Any]:
    return jsonify(OPENAPI_SPEC)

@rolls.route("/rolls", methods=["POST"])
def make_roll() -> Tuple[Dict[str, Any], HTTPStatus, Dict[str, str]]:
    body = request.get_json(force=True)
    if set(body.keys()) != {"dice"}:
        raise BadRequest(f"Extra fields in {body!r}")
    try:
        n_dice = int(body["dice"])
    except ValueError as ex:
        raise BadRequest(f"Bad 'dice' value in {body!r}")

    dice = make_dice(n_dice)
    SESSIONS[dice.identifier] = dice
    current_app.logger.info(f"Rolled roll={dice!r}")

    headers = {
        "Location":
        url_for("rolls.get_roll", identifier=dice.identifier)
    }
    return jsonify(asdict(dice)), HTTPStatus.CREATED, headers
```

The `SESSIONS` object uses an all-caps name to show that it is a global module. This

can become our database. Currently, it's initialized as an empty dictionary object. In a more sophisticated application, it could be a `shelve` object. Pragmatically, this would be replaced with a proper database driver to handle locking and concurrent access.

The `Blueprint` object, like the `Flask` object, is assigned a name. In many cases, the module name can be used. For this example, the module name will be used by the overall Flask application, so the name `"rolls"` was used.

In this example, we've assigned the `Blueprint` object to a `rolls` variable. It's also common to see a name such as `bp` used for this. A shorter name makes it slightly easier to type the route decorator name.

The preceding example defines two routes:

- One route is for the Open API Specification document.
- The second route is for the `/rolls` path, when used with the `POST` method.

This combination will lead to the typical four-part process of handling a RESTful request:

- **Parsing:** The Flask `request` object is interrogated to get the body of the request. The `request.get_json(force=True)` will use a JSON parser on the body of the request, irrespective of `Content-Type` header. While a well-written client application should provide appropriate headers, this code will tolerate a missing or incorrect header. A few validity checks are applied to the body of the request. In this example, a specialized exception is raised if the request doesn't include the `'dice'` key, and if the value for the `'dice'` key isn't a valid integer. More checks for valid inputs might include being sure the number of dice was between 1 and 100; this can help to avoid the problem of rolling millions or billions of dice. In the following example, we'll see how this exception is mapped to a proper HTTP status code of `400` Bad Request.
- **Evaluating:** The `make_dice()` function creates a `Dice` instance. This is saved into the global module `SESSIONS` database. Because each `Dice` instance is given a unique, randomized key, the `SESSIONS` database retains a history of all of the `Dice` objects. This is a simple in-memory database, and the information only lasts as long as the server is running. A more complete application would use a separate persistence layer.

- **Logging**. The Flask `current_app` object is used to get access to the Flask logger and write a log message on the response.
- **Responding**. The Flask `jsonify()` function is used to serialize the response into JSON Notation. This includes an additional `Location` header. This additional header is offered to a client to provide the correct, canonical URI for locating the object that was created. We've used the Flask `url_for()` function to create the proper URL. This function uses the name of the function, qualified by the `"rolls.get_roll"` blueprint name; the URI path is reverse-engineered from the function and the argument value.

The function for retrieving a roll uses a `GET` request, and is as follows:

```
@rolls.route("/rolls/<identifier>", methods=["GET"])
def get_roll(identifier) -> Tuple[Dict[str, Any], HTTPStatus]:
    if identifier not in SESSIONS:
        abort(HTTPStatus.NOT_FOUND)

    return jsonify(asdict(SESSIONS[identifier])), HTTPStatus.OK
```

This function represents a minimalist approach. If the identifier is unknown, a `404 Not Found` message is returned. There's no additional evaluation or logging, since no state change occurred. This function serializes the current state of the dice.

The response to a `PATCH` request is as follows:

```
@rolls.route("/rolls/<identifier>", methods=["PATCH"])
def patch_roll(identifier) -> Tuple[Dict[str, Any], HTTPStatus]:
    if identifier not in SESSIONS:
        abort(HTTPStatus.NOT_FOUND)
    body = request.get_json(force=True)
    if set(body.keys()) != {"keep"}:
        raise BadRequest(f"Extra fields in {body!r}")
    try:
        keep_positions = [int(d) for d in body["keep"]]
    except ValueError as ex:
        raise BadRequest(f"Bad 'keep' value in {body!r}")

    dice = SESSIONS[identifier]
    dice.reroll(keep_positions)

    return jsonify(asdict(dice)), HTTPStatus.OK
```

This has two kinds of validation to perform:

- The identifier must be valid.
- The document provided in the body of the request must also be valid.

This example performs a minimal validation to develop a list of positions assigned to the `keep_positions` variable.

The evaluation is performed by the `reroll()` method of the `Dice` instance. This fits the idealized notion of separating the `Dice` model from the RESTful application that exposes the model. There's relatively little substantial state-change processing in the Flask application.

Consider a desire for additional validation to confirm that the position values are between zero and the number of dice in `SESSIONS[identifier].roll`. Is this a part of the Flask application? Or, is this the responsibility of the `Dice` class definition?

There are two kinds of validations involved here:

- **Serialization syntax**: This validation includes a JSON syntax, a data type representation, and the isolated constraints expressed in the Open API Specification. This validation is limited to simple constraints on the values that can be serialized in JSON. We can formalize the `{"dice": d}` document schema to ensure that `k` is a positive integer. Similarly, we can formalize the `{"keep": [k, k, k, ...]}` document schema to require the values for `k` to be positive integers. However, there's no way to express a relationship between the values for `n` and `k`. There's also no way to express the requirement that the values for `k` be unique.
- **Problem domain data**: This validation goes beyond what can be specified in the Open API specification. It is tied to the problem domain and requires the JSON serialization be valid. This can include actions such as validating relationships among objects, confirming that state changes are permitted, or checking more nuanced rules such as string formats for email addresses.

This leads to multiple tiers of RESTful request parsing and validation. Some features of the request are tied directly to serialization and simple data type questions. Other features of a request are part of the problem domain, and must be deferred to the classes that support the RESTful API.

Registering a blueprint

A Flask application must have the blueprints registered. In some cases, there will be multiple instances of a blueprint, each at a different point in the URI path. In our case, we only have a single instance of the blueprint at the root of the URI path.

In the preceding examples, the bad request data was signaled by raising a `BadRequest` exception. A Flask error handler can be defined to transform an exception object to a standardized RESTful response document in JSON format.

The example Flask application is shown here:

```
class BadRequest(Exception):
    pass

def make_app() -> Flask:

    app = Flask(__name__)

    @app.errorhandler(BadRequest)
    def error_message(ex) -> Tuple[Dict[str, Any], HTTPStatus]:
        current_app.logger.error(f"{ex.args}")
        return jsonify(status="Bad Request", message=ex.args), HTTPStatus.BAD_REQUEST

    app.register_blueprint(rolls)

    return app
```

The function name, `make_app()`, is expected by Flask. Using this standard name makes it easy to launch the application. It also makes it easier to integrate with products such as uWSGI or **Gunicorn**.

Once the `Flask` object, `app`, is created, the `@app.errorhandler` decorator can be used to map Python exceptions to generic responses. This technique will replace Flask's default HTML-based responses with JSON-based responses. The advantage of JSON-format responses is that they make the RESTful API provide more consistent behavior.

The `rolls` blueprint, defined in previous examples, is registered at the root of the URI path. This `make_app()` function can also be used to include configuration parameters. It's also common to see a sequence of `extension.init_app(app)` function

calls for any of the Flask extensions being used.

Let's look at how we can create a secure REST service in the next section.

Creating a secure REST service

We can decompose application security into two considerations: authentication and authorization. We need to authenticate who the user is and we also need to be sure that the user is authorized to execute a particular function. There are a variety of techniques available for offering a secure RESTful service. All of them depend on using SSL. It's essential to create proper certificates and use them to ensure that all data transmissions are encrypted.

The details of setting up a certificate for SSL encryption are outside the scope of this book. The OpenSSL toolkit can be used to create self-signed certificates. The Flask application can then use these certificates as part of a testing or development environment.

When HTTP over SSL (HTTPS) is used, then the handling of credentials and authentication can be simplified. Without HTTPS, the credentials must be encrypted in some way to be sure they aren't *sniffed* in transmission. With HTTPS, credentials such as usernames and passwords can be included in headers by using a simple base-64 serialization of the bytes.

There are a number of ways of handling authentication; here are two common techniques:

- The HTTP `Authorization` header can be used (the header's purpose is authentication, but it's called `Authorization`). The `Basic` type is used to provide username and password. The `Bearer` type can be used to provide an `OAuth` token. This is often used by the presentation layer where a human supplies their password.
- The HTTP `Api-Key` header can be used to provide a role or authorization. This is often provided by an API client application, and is configured by trusted administrators. This can involve relatively simple authorization checks.

We'll show the simpler approach of working with internal `Api-Key` headers first. It's best to use a library such as `https://authlib.org` for dealing with user credentials and the `Authorization` header. While the essential rules for handling

passwords are generally simple, it's easier to use a well-respected package.

Using an `Api-Key` header for authorization checks is something that's designed to fit within the Flask framework. The general approach requires three elements:

- A repository of valid `Api-Key` values, or an algorithm for validating a key.
- An `init_app()` function to do any one-time preparation. This might include reading a file, or opening a database.
- A decorator for the application view functions.

Each of these elements is shown in the following example:

```
from functools import wraps
from typing import Callable, Set

VALID_API_KEYS: Set[str] = set()

def init_app(app):
    global VALID_API_KEYS
    if app.env == "development":
        VALID_API_KEYS = {"read-only", "admin", "write"}
    else:
        app.logger.info("Loading from {app.config['VALID_KEYS']}")
        raw_lines = (
            Path(app.config['VALID_KEYS'])
                .read_text()
                .splitlines()
        )
        VALID_API_KEYS = set(filter(None, raw_lines))

def valid_api_key(view_function: Callable) -> Callable:

    @wraps(view_function)
    def confirming_view_function(*args, **kw):
        api_key = request.headers.get("Api-Key")
        if api_key not in VALID_API_KEYS:
            current_app.logger.error(f"Rejecting Api-Key:{api_key!r}")
            abort(HTTPStatus.UNAUTHORIZED)
        return view_function(*args, **kw)

    return confirming_view_function
```

`VALID_API_KEYS` is a global module that contains the current set of valid values for the `Api-Keys` header. It has a type hint suggesting that it is a set of string values. We're using this instead of a more formal database. In many cases, this is all that's required to confirm that the `Api-Key` values sent by a client application meets the authorization requirement.

The `init_app()` function will load this global module from a file. For development, a simple set of `Api-Key` values are provided. This allows simple unit tests to

proceed.

`valid_api_key()` is a decorator. It's used to make a consistent check that can be used in all view function definitions to be sure that an `Api-Key` header is present and the value is known. It's a common practice to use the Flask `g` object to save this information for use in later logging messages. We might also add a line such as `g.api_key = api_key` in this function to save the value for the duration of the interaction.

Here's how we would modify a view function to use this new decorator:

```
@roll.route("/roll/<identifier>", methods=["GET"])
@valid_api_key
def get_roll(identifier) -> Tuple[Dict[str, Any], HTTPStatus]:
    if identifier not in SESSIONS:
        abort(HTTPStatus.NOT_FOUND)

    return jsonify(
        roll=SESSIONS[identifier], identifier=identifier, status="OK"
    ), HTTPStatus.OK
```

This is a copy of the `get_roll()` function shown in an earlier example. The change is to add the `@valid_api_key` authorization decorator after the route decorator. This provides assurance that only requests with an `Api-Key` header in the short list of valid keys will be able to retrieve the `roll` values.

A parameterized version of the `@valid_api_key` decorator is a bit more complex.

Hashing user passwords will be discussed in the next section.

Hashing user passwords

Perhaps the most important advice that can possibly be offered on the subject of security is the following:

> ***Never Store Passwords***
> *Only a repeated cryptographic hash of password plus salt can be stored on a server. The password itself must be utterly unrecoverable. Do not ever store any recoverable password as part of an application.*

Here's an example class that shows us how salted password hashing might work:

```
from hashlib import sha256
import os

class Authentication:
    iterations = 1000

    def __init__(self, username: bytes, password: bytes) -> None:
        """Works with bytes. Not Unicode strings."""
        self.username = username
        self.salt = os.urandom(24)
        self.hash = self._iter_hash(
            self.iterations, self.salt, username, password)

    @staticmethod
    def _iter_hash(iterations: int, salt: bytes, username: bytes, password: bytes):
        seed = salt + b":" + username + b":" + password
        for i in range(iterations):
            seed = sha256(seed).digest()
        return seed

    def __eq__(self, other: Any) -> bool:
        other = cast("Authentication", other)
        return self.username == other.username and self.hash == other.hash

    def __hash__(self) -> int:
        return hash(self.hash)

    def __repr__(self) -> str:
        salt_x = "".join("{0:x}".format(b) for b in self.salt)
        hash_x = "".join("{0:x}".format(b) for b in self.hash)
        return f"{self.username} {self.iterations:d}:{salt_x}:{hash_x}"

    def match(self, password: bytes) -> bool:
        test = self._iter_hash(
            self.iterations, self.salt,
            self.username, password)
        return self.hash == test  # Constant Time is Best
```

This class defines an `Authentication` object for a given username. The object contains the username, a unique random salt created each time the password is

set or reset, and the final hash of the salt plus the password.

This class also defines a `match()` method that will determine whether a given password will produce the same hash as the original password.

There are several ways to transform bytes to strings. The `"".join("{0:x}".format(b) for b in self.salt)` expression transforms the `salt` value into a string of hexadecimal digit pairs. Here's an example showing the result of the transformation.

```
>>> salt = b'salt'
>>> "".join("{0:x}".format(b) for b in salt)
'73616c74'
```

Note that no password is stored by this class. Only hashes of passwords are retained. We provided a comment (`# Constant Time is Best`) on the comparison function as a note that this implementation is incomplete. An algorithm that runs in constant time and isn't particularly fast is better than the built-in equality comparison among strings.

We also included a hash computation to emphasize that this object is immutable. An additional design feature would be to consider using `__slots__` to save storage.

Note that these algorithms work with byte strings, not Unicode strings. We either need to work with bytes or we need to work with the ASCII encoding of a Unicode username or password.

Let's see how to implement REST with a web application framework.

Implementing REST with a web application framework

A RESTful web server is a web application. This means we can leverage any of the popular Python web application frameworks. The complexity of creating RESTful services is low. The preceding examples illustrate how simple it is to map the CRUD rules to HTTP Methods.

Some of the Python web frameworks include one or more REST components. In some cases, the RESTful features are almost entirely built-in. In other cases, an add-on project can help define RESTful web services with minimal programming. For example, `https://flask-restful.readthedocs.io/en/latest/` is a REST framework that can be used as a Flask extension.

Searching **PyPI** (`https://pypi.python.org`) for REST will turn up a large number of packages. There are numerous solutions that are already available. Most of these will offer some levels of simplification for common cases.

A list of Python web frameworks can be found at `https://wiki.python.org/moin/WebFrameworks`. The point of these projects is to provide a reasonably complete environment to build web applications. Any of these can be used in place of Flask to create RESTful web services.

There are several compelling reasons for writing a RESTful service directly in Flask, and some are listed here:

- The REST service views are generally short and involve the minimal overheads required to deserialize JSON requests and serialize JSON responses. The remainder of the processing is unique to the problem domain.
- If there is unique processing that does not map to a REST package in an obvious way, it may be better to simply write the view functions. In particular, validation rules and state transition rules are often difficult to standardize and may be difficult to integrate with the built-in assumptions behind an existing REST package.

- If there's an unusual or atypical access layer or persistence mechanism, many REST packages have default assumptions about database operations, and are often tailored toward a package such as SQLAlchemy; for example, `https://flask-restless.readthedocs.io/en/stable/`.

Security, in particular, can be challenging. There are several best practices including the following:

- Always use SSL. For final production use, purchase a certificate from a trusted **Certification Authority** (**CA**).
- Never encrypt or store passwords, always use salted hashing.
- Avoid building home-brewed authentication. Use project such as `https://flask-dance.readthedocs.io/en/latest/` or `https://authlib.org`.

For the final production deployment, there are a number of alternatives. See `http://flask.pocoo.org/docs/1.0/deploying/` for a long list of the ways that Flask applications can be deployed at scale.

Let's see how to use a message queue to transmit objects.

Using a message queue to transmit objects

The `multiprocessing` module uses both the serialization and transmission of objects. We can use queues and pipes to serialize objects that are then transmitted to other processes. There are numerous external projects to provide sophisticated message queue processing. We'll focus on the `multiprocessing` queue because it's built in to Python and works nicely.

For high-performance applications, a faster message queue may be necessary. It may also be necessary to use a faster serialization technique than pickling. For this chapter, we'll focus only on the Python design issues. The multiprocessing module relies on `pickle` to encode objects. See `Chapter 10`, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML*, for more information. We can't provide a restricted unpickler easily; therefore, this module offers us some relatively simple security measures put into place to prevent any unpickle problems.

There is one important design consideration when using `multiprocessing`: it's generally best to avoid having multiple processes (or multiple threads) attempting to update shared objects. The synchronization and locking issues are so profound (and easy to get wrong) that the standard joke goes as follows:

When confronted with a problem, the programmer thinks, "I'll use multiple threads."

problems Now. two programmer the has

It's very easy for locking and buffering to make a mess of multithreaded processing.

Using process-level synchronization via RESTful web services or `multiprocessing` can prevent synchronization issues because there are no shared objects. The essential design principle is to look at the processing as a pipeline of discrete steps. Each processing step will have an input queue and an output queue; the step will fetch an object, perform some processing, and write the object.

The `multiprocessing` philosophy matches the `POSIX` concept of a shell pipeline,

written as `process1 | process2 | process3`. This kind of shell pipeline involves three concurrent processes interconnected with pipes. The important difference is that we don't need to use `STDIN`, `STDOUT`, or an explicit serialization of the objects. We can trust the `multiprocessing` module to handle the **operating system** (**OS**)-level infrastructure.

The POSIX shell pipelines are limited in that each pipe has a single producer and a single consumer. The Python `multiprocessing` module allows us to create message queues that include multiple consumers. This allows us to have a pipeline that fans out from one source process to multiple sink processes. A queue can also have multiple consumers that allow us to build a pipeline where the results of multiple source processes can be combined by a single sink process.

To maximize throughput on a given computer system, we need to have enough work pending so that no processor or core is ever left with nothing useful to do. When any given OS process is waiting for a resource, at least one other process should be ready to run.

When looking at our simulations, for example, we need to gather statistically significant simulation data by exercising a player strategy or betting strategy (or both) a number of times. The idea is to create a queue of processing requests so that our computer's processors (and cores) are fully engaged in processing our simulations.

Each processing request can be a Python object. The `multiprocessing` module will pickle that object so that it is transmitted via the queue to another process.

We'll revisit this in `Chapter 16`, *The Logging and Warning Modules*, when we look at how the `logging` module can use `multiprocessing` queues to provide a single, centralized log for separate producer processes. In these examples, the objects transmitted from process to process will be the `logging.LogRecord` instances.

In the next section, we'll learn how to define processes.

Defining processes

We must design each processing step as a `simple` loop that gets a request from a queue, processes that request, and places the results into another queue. This decomposes the larger problem into a number of stages that form a pipeline. Because each of these stages runs concurrently, the system resource use will be maximized. Furthermore, as the stages involve simple gets and puts into independent queues, there are no problems with complex locking or shared resources. A process can be a simple function or a callable object. We'll focus on defining processes as subclasses of `multiprocessing.Process`. This gives us the most flexibility.

For the simulation of a stateful process such as a game, we can break the simulation down into a three-step pipeline:

1. An overall driver puts simulation requests into a processing queue.
2. A pool of simulators will get a request from the processing queue, perform the simulation, and put the statistics into a results queue.
3. A summarizer will get the results from the result queue and create a final tabulation of the results.

Using a process pool allows us to have as many simulations running concurrently as our CPU can handle. The pool of simulators can be configured to ensure that simulations run as quickly as possible.

Here's a definition of the simulator process:

```
import multiprocessing

class Simulation(multiprocessing.Process):

    def __init__(
        self,
        setup_queue: multiprocessing.SimpleQueue,
        result_queue: multiprocessing.SimpleQueue,
    ) -> None:
        self.setup_queue = setup_queue
        self.result_queue = result_queue
        super().__init__()

    def run(self) -> None:
        """Waits for a termination"""
```

```
        print(f"{self.__class__.__name__} start")
        item = self.setup_queue.get()
        while item != (None, None):
            table, player = item
            self.sim = Simulate(table, player, samples=1)
            results = list(self.sim)
            self.result_queue.put((table, player, results[0]))
            item = self.setup_queue.get()
        print(f"{self.__class__.__name__} finish")
```

We've extended `multiprocessing.Process`. This means that we must do two things to work properly with multiprocessing: we must ensure that `super().__init__()` is executed, and we must override `run()`.

Within the body of `run()`, we're using two queues. `setup_queue` will contain two-tuples of the `Table` and `Player` objects. The process will use these two objects to run a simulation. It will put the resulting three-tuple into `result_queue`. The API for the `Simulate` class is this:

```
class Simulate:

    def __init__(
            self,
            table: Table,
            player: Player,
            samples: int
    ) -> None: ...

    def __iter__(self) -> Iterator[Tuple]: ...
```

The iterator will yield the requested number, `samples`, of statistical summaries. We've included a provision for a **sentinel object** to arrive via `setup_queue`. This object will be used to gracefully close down the processing. If we don't use a sentinel object, we'll be forced to terminate the processes, which can disrupt locks and other system resources. Here's the summarization process:

```
class Summarize(multiprocessing.Process):

    def __init__(self, queue: multiprocessing.SimpleQueue) -> None:
        self.queue = queue
        super().__init__()

    def run(self) -> None:
        """Waits for a termination"""
        print(f"{self.__class__.__name__} start")
        count = 0
        item = self.queue.get()
        while item != (None, None, None):
            print(item)
            count += 1
            item = self.queue.get()
        print(f"{self.__class__.__name__} finish {count}")
```

This also extends `multiprocessing.Process`. In this case, we're fetching items from a queue and simply counting them. A more useful processing might use several `collection.Counter` objects to accumulate more interesting statistics.

As with the `Simulation` class, we're also going to detect a sentinel and gracefully close down the processing. The use of a sentinel object allows us to close down processing as soon as the work is completed by the process. In some applications, the child process can be left running indefinitely.

Let's see how to build queues and supply data in the next section.

Building queues and supplying data

Objects are transferred among processes through queues or pipes. Building queues involves creating instances of `multiprocessing.Queue` or one of its subclasses. For this example, we can use the following:

```
setup_q = multiprocessing.SimpleQueue()
results_q = multiprocessing.SimpleQueue()
```

We created two queues that define the processing pipeline. When we put a simulation request into `setup_q`, we expect that a `Simulation` process will get the request pair and run the simulation. This should generate a results triple, which is put in `results_q`. The results triple should, in turn, lead the work being done by the `Summarize` process. Here's how we can start a single `Summarize` process:

```
result = Summarize(results_q)
result.start()
```

Here's how we can create four concurrent simulation processes:

```
    simulators = []
    for i in range(4):
        sim = Simulation(setup_q, results_q)
        sim.start()
        simulators.append(sim)
```

The four concurrent simulators will be competing for work. Each one will be attempting to grab the next request from the queue of pending requests. Once all the four simulators are busy working, the queue will start to get filled with unprocessed requests. The ideal size of the pool of workers is difficult to predict. It depends on the number of cores a processor has, and it also depends on the workload. A pool of processors doing a great deal of **input and output** (**I/O**) work will also do a lot of waiting for I/O to complete; so, the pool can be very large. On a smaller machine with only a four cores and a compute-intensive workload, the pool will be smaller.

After the queues and processes are waiting for work, the driver function can start putting requests into the `setup_q` queue. Here's a loop that will generate a flood of requests:

```
table = Table(
    decks=6, limit=50, dealer=Hit17(), split=ReSplit(),
    payout=(3, 2)
)
for bet in Flat, Martingale, OneThreeTwoSix:
    player = Player(SomeStrategy(), bet(), 100, 25)
    for sample in range(5):
        setup_q.put((table, player))
```

We have created a `Table` object. For each of the three betting strategies, we have created a `Player` object, and then queued up a simulation request. The pickled two-tuple will be fetched from the queue by the `Simulation` object and then it will be processed. In order to have an orderly termination, we'll need to queue sentinel objects for each simulator:

```
for sim in simulators:
    setup_q.put((None, None))

# Wait for the simulations to all finish.
for sim in simulators:
    sim.join()
```

We put a sentinel object into the queue for each simulator to consume. Once all the simulators have consumed the sentinels, we can wait for the processes to finish execution and join back into the parent process.

Once the `Process.join()` operation is finished, no more simulation data will be created. We can enqueue a sentinel object into the simulation results queue, as well:

```
results_q.put((None, None, None))
result.join()
```

Once the results sentinel object is processed, the `Summarize` process will stop accepting input and we can `join()` it as well.

We used multiprocessing to transmit objects from one process to another. This gives us a relatively simple way to create high-performance, multi-processing data pipelines. The `multiprocessing` module uses `pickle`, so there are few limitations on the nature of objects that can be pushed through the pipelines.

It's informative to adjust the pool size to see the impact of more workers and fewer workers on the elapsed runtime. The interaction among processing cores, memory, and the nature of the workload is difficult to predict, and an empirical study is helpful to find the optimum number of workers in the pool.

Summary

We have looked at transmitting and sharing objects using RESTful web services. RESTful web services via Flask are one way to do this. The `multiprocessing` module embodies a radically different solution to the same problem. Both of these architectures provide for communicating a representation of an object's state. In the case of `multiprocessing`, `pickle` is used to represent the state. In the case of building RESTful web services, we have to choose the representation(s) used. In the examples used here, we focused on JSON, because it's widely used and has a simple implementation.

RESTful web services often use a framework to simplify the code required for the standard features of HTTP request and response processing. In the examples, we identified four steps: parsing, evaluating, logging, and responding. The Flask framework provides a simple, extensible framework for this. Because WSGI applications have a simple, standardized API, we can easily a variety of frameworks for web applications. In this chapter, we looked at using Flask for implementing RESTful services.

We also looked at using `multiprocessing` to enqueue and dequeue messages from shared queues. This works nicely for passing objects from process to process. The beauty of using interprocess message queues is that we can avoid the locking problems associated with concurrent updates to objects shared by multiple threads.

Design considerations and tradeoffs

When building applications that involve concurrency, we must decide what grain of objects to make available and how to identify those objects with sensible URIs. With larger objects, we can easily achieve ACID properties. However, we may also be uploading and downloading too much data for our application's use cases. In some cases, we'll need to provide alternative levels of access: large objects to support ACID properties, and small objects to allow rapid response when a client application wants a subset of the data.

To implement more localized processing, we can leverage the `multiprocessing` module. This is focused more on building high-performance processing pipelines within a trusted host or network of hosts.

In some cases, the two design patterns are combined such that a RESTful request is handled by a multiprocessing pipeline. A conventional web server (such as NGINX) working through uWSGI will use multiprocessing techniques to pass the request through a named pipe from the frontend to the uWSGI-compliant Python application backend.

Schema evolution

When working with a public-facing API for RESTful services, we have to address the schema evolution problem. If we change a class definition, how will we change the response messages? If the external RESTful API must change for compatibility with other programs, how do we upgrade the Python web services to support a changing API?

Often, we'll have to provide a major release version number as part of our API. This might be provided explicitly as part of the path, or implicitly via data fields included in the `POST`, `PUT`, and `DELETE` requests.

We need to distinguish between changes that don't alter the URI paths or responses, and changes that will alter a URI or response. Minor changes to functionality will not change a URI or the structure of a response.

Changes to the URIs or the structure of a response may break an existing application. These are major changes. One way to make an application work gracefully through schema upgrades is to include version numbers in the URI paths. For example, `/roulette_2/wheel/` specifically names the second release of the roulette server.

Application software layers

Because of the relative sophistication available when using `sqlite3`, our application software must become more properly layered. For a REST client, we might look at a software architecture with layers.

When we are building a RESTful server, the presentation layer becomes greatly simplified. It is pared down to the essential request-response processing. It parses URIs and responds with documents in JSON or XML (or some other representation.) This layer should be reduced to a thin RESTful facade over the lower-level features.

In some complex cases, the front-most application, as viewed by human users, involves data from several distinct sources. One easy way to integrate data from diverse sources is to wrap each source in a RESTful API. This provides us with a uniform interface over distinct sources of data. It also allows us to write applications that gather these diverse kinds of data in a uniform way.

Looking forward

In the next chapter, we'll use persistence techniques to handle configuration files. A file that's editable by humans is the primary requirement for the configuration data. If we use a well-known persistence module, then our application can parse and validate the configuration data with less programming on our part.

Configuration Files and Persistence

A configuration file is a form of object persistence. It contains a serialized, plain-text, editable representation of some default state for an application program. We'll expand on the serialization techniques shown in `Chapter 10`, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML* to create files that are specifically used for application configuration. The focus on plain text means that pickle representation will be excluded. Due to the relative complexity of application configurations, CSV files aren't often used for this purpose, either.

Before a user can make use of an editable plain-text configuration file, we must design our application to be configurable. This can often require careful consideration of dependencies and limitations. Additionally, we must define some kind of configuration object that our application will use. In many cases, a configuration will be based on default values; we might want to allow for system-wide defaults with user-specific overrides to those defaults. We'll explore six representations for the configuration data, as follows:

- INI files use a format that was pioneered as part of Windows. The file is popular, in part, because it is an incumbent among available formats and has a long history.
- PY files are plain-old Python code. They have numerous advantages because of the familiarity and simplicity of the syntax. The configuration can be used by an `import` statement in the application.
- JSON and YAML are both designed to be user-friendly and easy to edit.
- Properties files are often used in a Java environment. They're relatively easy to work with and they are also designed to be human-friendly. There's no built-in parser for this, and this chapter includes regular expressions for this file format.
- XML files are popular but they are wordy, which means that, sometimes, they are difficult to edit properly. macOS uses an XML-based format called a property list or a PLIST file.

Each of these forms offers us some advantages and some disadvantages; there's no single technique that can be described as the best. In many cases, the choice is based on familiarity and compatibility with other software. In `Chapter`

15, *Design Principles and Patterns,* we'll return to this topic of configuration. Additionally, in later chapters, we'll make extensive use of configuration options.

In this chapter, we will cover the following topics:

- Configuration file use cases
- Representation, persistence, state, and usability
- Storing configurations in INI files and PY files
- Handling more literals via the `eval()` variant
- Storing configurations in PY files
- Why `exec()` is a non-problem
- Using `ChainMap` for defaults and overrides
- Storing configurations in JSON or YAML
- Using XML files such as PLIST and others

Technical requirements

The code files for this chapter are available at `https://git.io/fj2UP`.

Configuration file use cases

There are two configuration file use cases. Sometimes, we can stretch the definition slightly to add a third use case. The first two should be pretty clear:

- A person needs to edit a configuration file
- A piece of software will read a configuration file and make use of the options and arguments to tailor its behavior

Configuration files are rarely the *primary* input of an application program. Generally, they only tailor the program's behavior. For example, a web server's configuration file might tailor the behavior of the server, but the web requests are one primary input, and the database or filesystem is the other primary input. In the case of a GUI application, the user's interactive events are one input, and the files or database may be another input; a configuration file may fine-tune the application. The most notable exception to this general pattern is a simulation application; in this case, the configuration parameters might be the primary input.

There's also a blurry edge to the distinction between the application program and configuration input. Ideally, an application has one behavior irrespective of the configuration details. Pragmatically, however, the configuration might introduce additional strategies or states to an existing application, changing its behavior in fundamental ways. In this case, the configuration parameters become part of the code, not merely options or limits applied to a fixed code base.

In addition to being changed by a person, another use case for a configuration file is to save an application's current state. For example, in GUI applications, it's common to save the locations and sizes of various windows. As another example, a web interface often uses cookies to save the transaction state. It's generally helpful to segregate slowly-changing user-oriented configuration from the more dynamic attributes of the application's current operating state. The same types of file formats can be used, but the two kinds of uses are distinct.

A configuration file can provide a number of domains of arguments and parameter values to an application. We need to explore some of these various

kinds of data in more detail in order to decide how to represent them best. Some common types of parameters are listed as follows:

- Device names, which may overlap with the filesystem's location
- Filesystem locations and search paths
- Limits and boundaries
- Message templates and data format specifications
- Message text, possibly translated for internationalization
- Network names, addresses, and port numbers
- Optional behaviors, which are sometimes called feature toggles
- Security keys, tokens, usernames, and passwords

These values come from a number of distinct domains. Often, the configuration values have relatively common representations using strings, integers, and floating-point numbers. The intent is to use a tidy textual representation that's relatively easy for a person to edit. This means our Python applications must parse the human-oriented input.

In some cases, we may have lists of values where separators are required. We may also have large, multiline blocks of text. In these cases, the representation of the values may involve more complex punctuation and more complex parsing algorithms.

There is one additional configuration value that isn't a simple type with a tidy text representation. We could add this bullet to the preceding list:

- Additional features, plugins, and extensions; in effect, additional code

This kind of configuration value is challenging. We're not necessarily providing a simple string input or numeric limitation. In fact, we're providing the code to extend the application. When the plugin is in Python code, one option is to provide the path to an installed Python module as it would be used in an `import` statement using this dotted name: `package.module.object`. An application can then perform a variation of `from package.module import object` and use the given class or function as part of the application.

For a configuration that introduces non-Python code as part of a plugin or extension, we have two other techniques to make the external code usable:

- For binaries that aren't proper executable programs, we can try using the `ctypes` module to call defined API methods
- For binaries that are executable programs, the `subprocess` module gives us ways to execute them

Both of these techniques are beyond the scope of this book. This chapter will focus on the core issue of getting the arguments or the parameter values, which are conventional Python values.

Representation, persistence, state, and usability are discussed in the next section.

Representation, persistence, state, and usability

When looking at a configuration file, we're looking at a human-friendly version of an object state. Often, we'll provide the state of more than one object. When we edit a configuration file, we're changing the persistent state of an object that will get reloaded when the application is started (or restarted). We have two common ways of looking at a configuration file:

- A mapping or a group of mappings from parameter names to configuration values. Note that even when there are nested mappings, the structure is essentially keys and values.
- A serialized object that has complex attributes and properties with the configuration values. The distinguishing feature is the possibility of properties, methods, and derived values in addition to the user-supplied values.

Both of these views are equivalent; the mapping view relies on a built-in dictionary or namespace object. The serialized object will be a more complex Python object, which has been created from an external, human editable representation of the object. The advantage of a dictionary is the simplicity of putting a few parameters into a simple structure. The advantage of a serialized object is its ability to track the number of complex relationships.

For a flat dictionary or namespace to work, the parameter names must be chosen carefully. Part of designing the configuration is to design useful keys, which is something we looked at in `Chapter 11`, *Storing and Retrieving Objects via Shelve*, and `Chapter 12`, *Storing and Retrieving Objects via SQLite.* A mapping requires unique names so that other parts of the application can refer to it properly.

When we try to reduce a configuration file to a single mapping, we often discover that there are groups of related parameters. This leads to namespaces within the overall collection of names. Let's consider a web application that uses other web services; we might have two parallel groups of parameters: `service_one_host_name` and `service_one_port_number`, as well as `service_two_host_name` and

`service_two_port_number`. These could be four separate names, or we could use a more complex structure to combine the names into two related groups; for example, by perhaps creating a configuration data structure such as `{"service_one": {"host_name": "example.com", "port_number": 8080}, etc.}`.

There is a blurry space between using simple mappings and using more complex serialized Python objects. Some of the modules we'll look at in this chapter use complex nested dictionaries and namespace objects. The variety of alternative solutions suggests there is no single *best* way to organize the configuration parameters.

It's helpful to look at the `logging` configuration for examples of how it can be very challenging to configure a complex system. The relationships between Python logging object-loggers, formatters, filters, and handlers must all be bound together to create a logger usable by the application. If any pieces are missing, the logger will not produce output. Section 16.8 of the *Standard Library Reference* describes two different syntaxes for logging configuration files. We'll look at logging in `Chapter 16`, *The Logging and Warning Modules*.

In some cases, it may be simpler to use Python code directly as the configuration file. In this case, the configuration is a Python module, and the details are brought in using a simple `import` statement. If a configuration file's syntax adds too much complexity, then it may not be of any real value.

Once we've decided on the overall data structure, there are two common design patterns for the scope of that structure.

Application configuration design patterns

There are two core design patterns for the scope of objects used to configure a Python application:

- **A global property map**: A global object can contain all of the configuration parameters. A simple class definition is perhaps an ideal way to provide names and values; this tends to follow the **Singleton** design pattern ensuring that only one instance exists. Alternatives include a dictionary with pairs of `name: value`, or a `types.SimpleNamespace` object of attribute values.
- **Object construction**: Instead of a single object, we'll define a kind of **Factory** or collection of **Factories** that use the configuration data to build the objects of the application. In this case, the configuration information is used once when a program is started and never again. The configuration information isn't kept around as a global object.

The global property map design is very popular because it is simple and extensible. The first example of this is using a class object, which is defined as follows:

```
class Configuration:
    some_attribute = "default_value"
```

We can use the preceding class definition as a global container of attributes. During the initialization, we might have something similar to this as part of parsing the configuration file:

```
Configuration.some_attribute = "user-supplied value"
```

Everywhere else in the program, we can use the value of `Configuration.some_attribute`. A variation on this theme is to make a more formal **Singleton** class design pattern. This is often done with a global module, as it can be easily imported in a way that provides us with an accessible global definition.

The second example involves using a module for the configuration. We might

have a module named `configuration.py`. In that file, we can have a definition like the following:

```
settings = {
    "some_attribute": "user-supplied value"
}
```

Now, the application can use `configuration.settings` as a global repository for all of the application's settings. A function or class can parse the configuration file, loading this dictionary with the configuration values that the application will then use.

In a Blackjack simulation, we might see the following code:

```
shoe = Deck(configuration.settings['decks'])
```

Alternatively, we might possibly see the following code:

```
If bet > configuration.settings['limit']: raise InvalidBet()
```

Generally, we'll try to avoid having a global variable for the configuration. Because a global variable is implicitly present everywhere, it can be misused to carry stateful processing in addition to configuration values. Instead of a global variable, we can often handle the configuration relatively more neatly through object construction. In the next section, we will look at examples of constructing objects as a way to implement the configuration changes.

Configuring via object construction

When configuring an application through object construction, the objective is to build the required objects at startup time. In effect, the configuration file defines the various initialization parameters for the objects that will be built.

We can often centralize much of this initial object construction in a single `main()` function. This will create the objects that do the real work of the application. We'll revisit and expand on these design issues in `Chapter 18`, *Coping with the Command Line*.

Let's now consider a simulation of Blackjack playing and betting strategies. When we run a simulation, we want to gather the performance of a particular combination of independent variables. These variables might include some casino policies including the number of decks, table limits, and dealer rules. The variables might include the player's game strategies for when to hit, stand, split, and double down. It could also include the player's betting strategies of flat betting, Martingale betting, or some more Byzantine type of betting system; our baseline code starts out like this:

```
import csv

def simulate_blackjack() -> None:
    # Configuration
    dealer_rule = Hit17()
    split_rule = NoReSplitAces()
    table = Table(
        decks=6, limit=50, dealer=dealer_rule,
        split=split_rule, payout=(3, 2)
    )
    player_rule = SomeStrategy()
    betting_rule = Flat()
    player = Player(
        play=player_rule, betting=betting_rule,
        max_rounds=100, init_stake=50
    )

    # Operation
    simulator = Simulate(table, player, samples=100)
    result_path = Path.cwd() / "data" / "ch14_simulation.dat"
    with result_path.open("w", newline="") as results:
        wtr = csv.writer(results)
        wtr.writerows(gamestats)
```

In this example, the `Configuration` part of the code builds the six individual objects

to be used in the `Operation` phase. These objects include `dealer_rule`, `split_rule`, `table`, `player_rule`, `betting_rule`, and `player`. Additionally, there is a complex set of dependencies between `table` and subsidiary objects as well as `player` and two other objects.

The second part of the code, `Operation`, builds a `Simulate` instance using `table` and `player`. A `csv` writer object then writes rows from the `simulator` instance. This final `writerows()` function depends on the `Simulate` class providing a `__next__()` method.

The preceding example is a kind of technology spike – an initial draft solution – with hardcoded object instances and initial values. Any change is essentially a rewrite. A more polished application will rely on externally-supplied configuration parameters to determine the classes of objects and their initial values. When we separate the configuration parameters from the code, it means we don't have to *tweak* the code to make a change. This gives us consistent, testable software. A small change is accomplished by changing the configuration inputs instead of changing the code.

The `Simulate` class has an API that is similar to the following code:

```
from dataclasses import dataclass

@dataclass
class Simulate:
    """Mock simulation."""

    table: Table
    player: Player
    samples: int

    def __iter__(self) -> Iterator[Tuple]:
        """Yield statistical samples."""
        # Actual processing goes here...
```

This allows us to build the `Simulate()` object with some appropriate initialization parameters. Once we've built an instance of `Simulate()`, we can iterate through that object to get a series of statistical summary objects.

The next version of this can use configuration parameters from a configuration file instead of the hardcoded class names. For example, a parameter should be used to decide whether to create an instance of `Hit17` or `Stand17` for the `dealer_rule` value. Similarly, the `split_rule` value should be a choice among several classes that embody several different split rules used in casinos.

In other cases, parameter values should be used to provide arguments to the `Simulate` class `__init__()` method. For example, the number of decks, the house betting limit, and the Blackjack payout values are configuration values used to create the `Table` instance.

Once the objects are built, they interact normally through the `Simulate.__next__()` method to produce a sequence of statistical output values. No further need of a global pool of parameters is required: the parameter values are bound into the objects through their instance variables.

The object construction design is not as simple as a global property map. While more complex, it has the advantage of avoiding a global variable, and it also has the advantage of making the parameter processing central and obvious in the main factory function.

Adding new parameters when using object construction may lead to refactoring the application to expose a parameter or a relationship. This can make it seem more complex than a global mapping from name to value.

One significant advantage of this technique is the removal of the complex `if` statements deep within the application. Using the `Strategy` design pattern tends to push decision-making forward into object construction. In addition to simplifying the processing, the elimination of the `if` statements means there are fewer statements to execute and this can lead to a performance boost.

In the next section, we will demonstrate how to implement a configuration hierarchy.

Implementing a configuration hierarchy

We often have several choices as to where a configuration file should be placed. There are several common locations, and we can use any combination of choices to create a kind of inheritance hierarchy for the parameters:

- **The Python installation directory**: We can find the installed location for a module using the `__file__` attribute of the module. From here, we can use a `Path` object to locate a configuration file:

```
>>> import this
>>> from pathlib import Path
>>> Path(this.__file__)
PosixPath('/Users/slott/miniconda3/envs/mastering/lib/python3.7/this.py')
```

- **The system application installation directory**: This is often based on an owning username. In some cases, we can simply create a special user ID to own the application itself. This lets us use `~theapp/` as a configuration location. We can use `Path("~theapp").expanduser()` to track down the configuration defaults. In other cases, the application's code may live in the `/opt` or `/var` directories.
- **A system-wide configuration directory**: This is often present in `/etc`. Note that this can be transformed into `C:\etc` on Windows.
- **The current user's home directory**: We generally use `Path.home()` to identify the user's home directory.
- **The current working directory**: We generally use `Path.cwd()` to identify the current working directory.
- **A file named in the command-line parameters**: This is an explicitly named file and no further processing should be done to the name.

An application can integrate configuration options from all of these sources. Any installation default values should be considered the most generic and least user-specific; these defaults can be overridden by more specific values.

This can lead to a list of files like the following code:

```
from pathlib import Path
```

```
config_locations = (
    Path(__file__),
    # Path("~thisapp").expanduser(), requires special username
    Path("/opt") / "someapp",
    Path("/etc") / "someapp",
    Path.home(),
    Path.cwd(),
)
candidates = (dir / "someapp.config"
     for dir in config_locations)
config_paths = [path for path in candidates if path.exists()]
```

Here, the `config_locations` variable is a tuple of alternative paths where a configuration file might be located. The `candidates` generator will create paths that include a base path with a common base name, `someapp.config`. A final list object, `config_paths`, is built for those paths that actually exist. The idea is to provide the most generic names first, and the most user-specific names last.

Once we have this list of configuration filenames, we can append any filename supplied through the command-line arguments to the end of the list with the following code:

```
config_paths.append(command_line_option)
```

This gives us a list of locations to place a user-updated configuration file as well as the configuration default values.

Let's take a look at how to store the configuration in INI files.

Storing the configuration in INI files

The INI file format has historical origins from early Windows OS. The module to parse these files is `configparser`. For additional details on the INI file, you can refer to this Wikipedia article for numerous useful links: http://en.wikipedia.org/wiki/INI_file.

An INI file has *sections* and *properties* within each section. Our sample main program has three sections: the table configuration, player configuration, and overall simulation data gathering. For this simulation, we will use an INI file that is similar to the following example:

```
; Default casino rules
[table]
    dealer= Hit17
    split= NoResplitAces
    decks= 6
    limit= 50
    payout= (3,2)

; Player with SomeStrategy
; Need to compare with OtherStrategy
[player]
    play= SomeStrategy
    betting= Flat
    max_rounds= 100
    init_stake= 50

[simulator]
    samples= 100
    outputfile= p2_c13_simulation.dat
```

We've broken the parameters into three sections. Within each section, we've provided some named parameters that correspond to the class names and initialization values shown in our preceding model application initialization.

A single file can be parsed with the code shown in this example:

```
import configparser
config = configparser.ConfigParser()
config.read('blackjack.ini')
```

Here, we've created an instance of the parser and provided the target configuration filename to that parser. The parser will read the file, locate the sections, and locate the individual properties within each section.

If we want to support multiple locations for files, we can use `config.read(config_names)`. When we provide the list of filenames to `ConfigParser.read()`, it will read the files in a particular order. We want to provide the files from the most generic first to the most specific last. The generic configuration files that are part of the software installation will be parsed first to provide defaults. The user-specific configuration will be parsed later to override these defaults.

Once we've parsed the file, we need to make use of the various parameters and settings. Here's a function that constructs our objects based on a given configuration object created by parsing the configuration files. We'll break this into three parts; here's the part that builds the `Table` instance:

```
def main_ini(config: configparser.ConfigParser) -> None:
    dealer_nm = config.get("table", "dealer", fallback="Hit17")
    dealer_rule = {
        "Hit17": Hit17(),
        "Stand17": Stand17(),
    }.get(dealer_nm, Hit17())
    split_nm = config.get("table", "split", fallback="ReSplit")
    split_rule = {
        "ReSplit": ReSplit(),
        "NoReSplit": NoReSplit(),
        "NoReSplitAces": NoReSplitAces(),
    }.get(split_nm, ReSplit())
    decks = config.getint("table", "decks", fallback=6)
    limit = config.getint("table", "limit", fallback=100)
    payout = eval(
        config.get("table", "payout", fallback="(3,2)")
    )
    table = Table(
        decks=decks, limit=limit, dealer=dealer_rule,
        split=split_rule, payout=payout
    )
```

We've used properties from the `[table]` section of the INI file to select class names and provide initialization values. There are three broad kinds of cases here:

- **Mapping a string to a class name**: We've used a mapping to look up an object based on a string class name. This was done to create `dealer_rule` and `split_rule`. If the pool of classes was subject to considerable change, we might move this mapping into a separate factory function. The `.get()` method of a dictionary includes a default object instance, for example, `Hit17()`.
- **Getting a value that `ConfigParser` can parse for us**: The class can directly handle values of built-in types such as `str`, `int`, `float`, and `bool`. Methods such

as `getint()` handle these conversions. The class has a sophisticated mapping from a string to a Boolean, using a wide variety of common codes and synonyms for `True` and `False`.
- **Evaluating something that's not built-in**: In the case of `payout`, we had a string value, `'(3,2)'`, that is not a directly supported data type for `ConfigParser`. We have two choices to handle this. We can try and parse it ourselves, or we can insist that the value be a valid Python expression and make Python do this. In this case, we've used `eval()`. Some programmers call this a *security problem*. The next section deals with this.

Here's the second part of this example, which uses properties from the `[player]` section of the INI file to select classes and argument values:

```
    player_nm = config.get(
        "player", "play", fallback="SomeStrategy")
    player_rule = {
        "SomeStrategy": SomeStrategy(),
        "AnotherStrategy": AnotherStrategy()
    }.get(player_nm, SomeStrategy())
    bet_nm = config.get("player", "betting", fallback="Flat")
    betting_rule = {
        "Flat": Flat(),
        "Martingale": Martingale(),
        "OneThreeTwoSix": OneThreeTwoSix()
    }.get(bet_nm, Flat())
    max_rounds = config.getint("player", "max_rounds", fallback=100)
    init_stake = config.getint("player", "init_stake", fallback=50)
    player = Player(
        play=player_rule,
        betting=betting_rule,
        max_rounds=max_rounds,
        init_stake=init_stake
    )
```

This uses string-to-class mapping as well as built-in data types. It initializes two strategy objects and then creates `Player` from those two strategies, plus two integer configuration values.

Here's the final part; this creates the overall simulator:

```
outputfile = config.get(
    "simulator", "outputfile", fallback="blackjack.csv")
samples = config.getint("simulator", "samples", fallback=100)
simulator = Simulate(table, player, samples=samples)
with Path(outputfile).open("w", newline="") as results:
    wtr = csv.writer(results)
    wtr.writerows(simulator)
```

We've used two parameters from the `[simulator]` section that are outside the narrow confines of object creation. The `outputfile` property is used to name a file;

the `samples` property is provided as an argument to a method function.

The next section demonstrates how to handle more literals through the `eval()` variants.

Handling more literals via the eval() variants

A configuration file may have values of types that don't have simple string representations. For example, a collection might be provided as a `tuple` or a `list` literal; a mapping might be provided as a `dict` literal. We have several choices to handle these more complex values.

The choices resolve the issue of how much Python syntax the conversion is able to tolerate. For some types (`int`, `float`, `bool`, `complex`, `decimal.Decimal`, and `fractions.Fraction`), we can safely convert the string to a literal value because the `__init__()` object for these types can handle string values.

For other types, however, we can't simply do the string conversion. We have several choices on how to proceed:

- Forbid these data types and rely on the configuration file syntax plus processing rules to assemble complex Python values from very simple parts; this is tedious but it can be made to work. In the case of the table payout, we need to break the payout into two separate configuration items for the numerator and denominator. This is a lot of configuration file complexity for a simple two-tuple.
- Use `ast.literal_eval()` as it handles many cases of Python literal values. This is often the ideal solution.
- Use `eval()` to simply evaluate the string and create the expected Python object. This will parse more kinds of objects than `ast.literal_eval()`. But, do consider whether this level of generality is really needed.
- Use the `ast` module to compile and then vet the resulting code object. This vetting process can check for the `import` statements as well as use some small set of permitted modules. This is quite complex; if we're effectively allowing code, perhaps we should be designing a framework and simply including Python code.

If we are performing RESTful transfers of Python objects through the network, `eval()` of the resulting text cannot be trusted. You can refer to `Chapter`

10, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML*.

In the case of reading a local configuration file, however, `eval()` is certainly usable. In some cases, the Python application code is as easily modified as the configuration file. Worrying about `eval()` may not be helpful when the base code can be tweaked.

Here's how we use `ast.literal_eval()` instead of `eval()`:

```
>>> import ast
>>> ast.literal_eval('(3,2)')
(3, 2)
```

This broadens the domain of possible values in a configuration file. It doesn't allow arbitrary Python objects, but it allows a broad spectrum of literal values including tuples.

Let's take a look at how to store the configuration in PY files.

Storing the configuration in PY files

The PY file format means using Python code as the configuration file. This pairs nicely with the use of the same language implementing the application. We will have a configuration file that's simply a module; the configuration is written in the Python syntax. This can remove the need for sophisticated parsing to get the configuration values.

Using Python gives us a number of design considerations. We have two overall strategies to use Python as the configuration file:

- A top-level script: In this case, the configuration file is simply the top-most main program.
- An `exec()` import: In this case, our configuration file provides parameter values that are collected into module global variables.

We can design a top-level script file that looks like the following code:

```
from simulator import *

def simulate_SomeStrategy_Flat() -> None:
    dealer_rule = Hit17()
    split_rule = NoReSplitAces()
    table = Table(
        decks=6, limit=50, dealer=dealer_rule, split=split_rule, payout=(3, 2)
    )
    player_rule = SomeStrategy()
    betting_rule = Flat()
    player = Player(
        play=player_rule, betting=betting_rule, max_rounds=100, init_stake=50)

    simulate(table, player, Path.cwd()/"data"/"ch14_simulation2a.dat", 100)

if __name__ == "__main__":
    simulate_SomeStrategy_Flat()
```

This presents a number of configuration parameters used to create and initialize objects. In this kind of application, the configuration is simply written as code. We've factored out the common processing into a separate function, `simulate()`, which uses the configured objects, `table` and `player`; the `Path` target; and the number of samples to generate. Rather than parsing and converting strings, the configuration is presented as code.

One potential disadvantage of using Python as the configuration language is the potential complexity of the Python syntax. This is usually an irrelevant problem for two reasons. First, with some careful design, the syntax of the configuration should be simple assignment statements with a few `()` and `,` instances. Second, and more importantly, other configuration files have their own complex syntax, which are distinct from the Python syntax. Using a single language is a net reduction in complexity.

The `simulate()` function is imported from the overall `simulator` application. This `simulate()` function is similar to the following code:

```
import csv
from pathlib import Path

def simulate(table: Table, player: Player, outputpath: Path, samples: int) -> None:
    simulator = Simulate(table, player, samples=samples)
    with outputpath.open("w", newline="") as results:
        wtr = csv.writer(results)
        for gamestats in simulator:
            wtr.writerow(gamestats)
```

This function is generic with respect to the table, player, filename, and number of samples. Given the required configuration objects, it builds the final `Simulate` instance and collects the resulting data.

A potential difficulty with this kind of configuration technique is the lack of handy default values. The top-level script must be complete, that is, *all* of the configuration parameters must be present. In most cases, this is not a *limitation*. In the few cases where default values are important, we'll look at two ways to provide helpful default values.

Configuration via class definitions

The difficulty that we sometimes have with top-level script configuration is the lack of handy default values. To provide defaults, we can use ordinary class inheritance. Here's how we can use the class definition to build an object with the configuration values:

```
class Example2(simulation.AppConfig):
    dealer_rule = Hit17()
    split_rule = NoReSplitAces()
    table = Table(
        decks=6, limit=50, dealer=dealer_rule, split=split_rule, payout=(3, 2)
    )
    player_rule = SomeStrategy()
    betting_rule = Flat()
    player = Player(play=player_rule, betting=betting_rule, max_rounds=100, init_stake=5
    outputfile = Path.cwd()/"data"/"ch14_simulation2b.dat"
    samples = 100
```

This allows us to define the `AppConfig` class with default configuration values. The class that we've defined here, `Example2`, can override the default values defined in the `AppConfig` class.

We can also use mixins to break the definition down into reusable pieces. We might break our classes down into the table, player, and simulation components and combine them via mixins. For more information on the mixin class design, see `Chapter 9`, *Decorators and Mixins – Cross-Cutting Aspects*.

In two small ways, this use of a class definition pushes the envelope on object-oriented design. This kind of class has no method definitions; we're only going to use this class as a **Singleton** object. However, it is a very tidy way of packing up a small block of code so that the assignment statements fill in a small namespace.

We can modify our `simulate()` function to accept this class definition as an argument:

```
def simulate_c(config: Union[Type[AppConfig], SimpleNamespace]) -> None:
    simulator = Simulate(config.table, config.player, config.samples)
    with Path(config.outputfile).open("w", newline="") as results:
        wtr = csv.writer(results)
        wtr.writerow(simulator)
```

This function has picked out the relevant values, `config.table`, `config.player`, and `config.samples`, from the overall configuration object and used them to build a `Simulate` instance and execute that instance. The results are the same as the previous `simulate()` function, but the argument structure is different. Here's how we provide the single instance of the class to this function:

```
if __name__ == "__main__":
    simulation.simulate_c(Example2)
```

Note that we're not providing an instance of the `Example2` class. We're using the class object itself. The `Type[AppConfig]` type hint shows the class itself is expected, not an instance of the class.

One potential disadvantage of this approach is that it is not compatible with `argparse` to gather command-line arguments. We can solve this by defining the interface to be compatible with a `types.SimpleNamespace` object. This overlap is formalized in the type hint: `Union[Type[AppConfig], SimpleNamespace]`. This type definition permits a wide variety of objects to be used to provide configuration parameters.

In addition to using a class, we can also create a `SimpleNamespace` object to have a similar-looking syntax for using the configuration parameter values.

Configuration via SimpleNamespace

Using a `types.SimpleNamespace` object allows us to simply add attributes as needed; this is similar to using a class definition. When defining a class, all of the assignment statements are localized to the class. When creating a `SimpleNamespace` object, we'll need to explicitly qualify every name with the `NameSpace` object that we're populating. Ideally, we can create `SimpleNamespace` like the following code:

```
>>> import types
>>> config = types.SimpleNamespace(
...     param1="some value",
...     param2=3.14,
... )
>>> config
namespace(param1='some value', param2=3.14)
```

This works delightfully well if all of the configuration values are independent of each other. In our case, however, we have some complex dependencies among the configuration values. We can handle this in one of the following two ways:

- Provide only the independent values and leave it to the application to build the dependent values
- Build the values in the namespace incrementally

To create only the independent values, we might do something like this:

```
import types
config2c = types.SimpleNamespace(
  dealer_rule=Hit17(),
  split_rule=NoReSplitAces(),
  player_rule=SomeStrategy(),
  betting_rule=Flat(),
  outputfile=Path.cwd()/"data"/"ch14_simulation2c.dat",
  samples=100,
  )
config2c.table = Table(
    decks=6,
    limit=50,
    dealer=config2c.dealer_rule,
    split=config2c.split_rule,
    payout=(3, 2),
)
config2c.player = Player(
    play=config2c.player_rule,
    betting=config2c.betting_rule,
    max_rounds=100,
    init_stake=50
)
```

Here, we created `SimpleNamespace` with the six independent values for the configuration. Then, we updated the configuration to add two more values that are dependent on four of the independent values.

The `config2c` object is nearly identical to the object that was created by evaluating `Example4()` in the preceding example. Note that the base class is different, but the set of attributes and their values are identical. Here's the alternative, where we build the configuration incrementally in a top-level script:

```
from types import SimpleNamespace

config2d = SimpleNamespace()
config2d.dealer_rule = Hit17()
config2d.split_rule = NoReSplitAces()
config2d.table = Table(
    decks=6,
    limit=50,
    dealer=config2d.dealer_rule,
    split=config2d.split_rule,
    payout=(3, 2),
)
config2d.player_rule = SomeStrategy()
config2d.betting_rule = Flat()
config2d.player = Player(
    play=config2d.player_rule,
    betting=config2d.betting_rule,
    max_rounds=100,
    init_stake=50
)
config2d.outputfile = Path.cwd() / "data" / "ch14_simulation2d.dat"
config2d.samples = 100
```

The same `simulate_c()` function shown previously can be used for this configuration object.

Sadly, this suffers from the same problem as configuration using a top-level script. There's no handy way to provide default values to a configuration object.

An easy way to provide defaults is through a function that includes default parameter values.

We might want to have a factory function that we can import, which creates `SimpleNamespace` with the appropriate default values:

```
From simulation import  make_config
config2 = make_config()
```

If we used something like the preceding code, then the `config2` object would have

the default values assigned by the factory function, `make_config()`. A user-supplied configuration only needs to provide overrides for the default values.

Our default-supplying `make_config()` function can use the following code:

```
def make_config(
    dealer_rule: DealerRule = Hit17(),
    split_rule: SplitRule = NoReSplitAces(),
    decks: int = 6,
    limit: int = 50,
    payout: Tuple[int, int] = (3, 2),
    player_rule: PlayerStrategy = SomeStrategy(),
    betting_rule: BettingStrategy = Flat(),
    base_name: str = "ch14_simulation2e.dat",
    samples: int = 100,
) -> SimpleNamespace:
    return SimpleNamespace(
        dealer_rule=dealer_rule,
        split_rule=split_rule,
        table=Table(
            decks=decks,
            limit=limit,
            dealer=dealer_rule,
            split=split_rule,
            payout=payout,
        ),
        payer_rule=player_rule,
        betting_rule=betting_rule,
        player=Player(
            play=player_rule,
            betting=betting_rule,
            max_rounds=100,
            init_stake=50
        ),
        outputfile=Path.cwd() / "data" / base_name,
        samples=samples,
    )
```

Here, the `make_config()` function will build a default configuration through a sequence of assignment statements. The derived configuration values, including the `table` attribute and the `player` attribute, are built from the original inputs.

An application can then set only the interesting *override* values, as follows:

```
config_b = make_config(dealer_rule=Stand17())
simulate_c(config_b)
```

This seems to maintain considerable clarity by specifying only the override values.

All of the techniques from `Chapter 2`, *The __init__() Method*, apply to the definition of this kind of configuration factory function. We can build in a great

deal of flexibility if we need to. This has the advantage of fitting nicely with the way that the `argparse` module parses command-line arguments. We'll expand on this in `Chapter 18`, *Coping with the Command Line*.

Let's explore how to use Python with `exec()` for configuration.

Using Python with exec() for the configuration

When we decide to use Python as the notation for a configuration, we can use the `exec()` function to evaluate a block of code in a constrained namespace. We can imagine writing configuration files that look like the following code:

```
# SomeStrategy setup

# Table
dealer_rule = Hit17()
split_rule = NoReSplitAces()
table = Table(decks=6, limit=50, dealer=dealer_rule,
        split=split_rule, payout=(3,2))

# Player
player_rule = SomeStrategy()
betting_rule = Flat()
player = Player(play=player_rule, betting=betting_rule,
        max_rounds=100, init_stake=50)

# Simulation
outputfile = Path.cwd()/"data"/"ch14_simulation3a.dat"
samples = 100
```

This is a pleasant, easy-to-read set of configuration parameters. It's similar to the INI file and properties file that we'll explore in the following section. We can evaluate this file, creating a kind of namespace, using the `exec()` function:

```
code = compile(py_file.read(), "stringio", "exec")
assignments: Dict[str, Any] = dict()
exec(code, globals(), assignments)
config = SimpleNamespace(**assignments)

simulate(config.table, config.player, config.outputfile, config.samples)
```

In this example, the code object, `code`, is created with the `compile()` function. Note that this isn't required; we can simply provide the text of the file to the `exec()` function and it will compile the code and execute it.

The call to `exec()` provides three arguments:

- The compiled code object
- A dictionary that should be used to resolve any global names

- A dictionary that will be used for any locals that get created

When the code block is finished, the assignment statements will have been used to build values in the local dictionary; in this case, the `assignments` variable. The keys will be the variable names. This is then transformed into a `SimpleNamespace` object so it's compatible with other initialization techniques mentioned previously.

The `assignments` dictionary will have a value that looks like the following output:

```
{'betting_rule': Flat(),
 'dealer_rule': Hit17(),
 'outputfile': PosixPath('/Users/slott/mastering-oo-python-2e/data/ch14_simulation3a.dat
 'player': Player(play=SomeStrategy(), betting=Flat(), max_rounds=100, init_stake=50, rc
 'player_rule': SomeStrategy(),
 'samples': 100,
 'split_rule': NoReSplitAces(),
 'table': Table(decks=6, limit=50, dealer=Hit17(), split=NoReSplitAces(), payout=(3, 2))
```

This is used to create a `SimpleNamespace` object, `config`. The namespace object can then be used by the `simulate()` function to perform the simulation. Using a `SimpleNamespace` object makes it easier to refer to the individual configuration settings. The initial dictionary requires code such as `assignments['samples']`. The resulting config object can be used with code such as `config.samples`.

The next section is a digression on why using `exec()` to parse Python code is not a security risk.

Why exec() is a non-problem

The previous section discussed `eval()`; the same considerations also apply to `exec()`.

Generally, the set of available `globals()` is tightly controlled. Access to the `os` and `subprocess` modules, or the `__import__()` function, can be eliminated by removing them from the globals provided to `exec()`.

If you have an evil programmer who will cleverly corrupt the configuration files, then recall that they have complete access to the entire Python source. So, why would they waste time cleverly tweaking configuration files when they can just change the application code itself?

One question can be summarized like this: *What if someone thinks they can monkey patch the application by forcing new code in via the configuration file?* The person trying this is just as likely to break the application through a number of other equally clever or deranged channels. Avoiding Python configuration files won't stop the unscrupulous programmer from breaking things by doing something else that's ill-advised. The Python source is directly available for modification, so unnecessarily worrying about `exec()` may not be beneficial.

In some cases, it may be necessary to change the philosophy. An application that's highly customizable might actually be a general framework, not a tidy, finished application. A framework is designed to be extended with additional code.

In the case where configuration parameters are downloaded through a web application, then `exec()`, `eval()`, and Python syntax should not be used. For these cases, the parameters need to be in a language such as JSON or YAML. Accepting a configuration file from a remote computer is a type of RESTful state transfer. This is covered in `Chapter 13`, *Transmitting and Sharing Objects*.

In the next section, we'll explore one of the collections as a way to provide override values and default values in a single, convenient object.

Using ChainMap for defaults and overrides

We'll often have a configuration file hierarchy. Previously, we listed several locations where configuration files can be installed. The `configparser` module, for example, is designed to read a number of files in a particular order and integrate the settings by having later files override values from earlier files.

We can implement elegant default-value processing using the `collections.ChainMap` class. You can refer to Chapter 7, *Creating Containers and Collections* for some background on this class. We'll need to keep the configuration parameters as `dict` instances, which is something that works out well using `exec()` to evaluate Python-language initialization files.

Using this will require us to design our configuration parameters as a flat dictionary of values. This may be a bit of a burden for applications with a large number of complex configuration values, which are integrated from several sources. We'll show you a sensible way to flatten the names.

First, we'll build a list of files based on the standard locations:

```
from collections import ChainMap
from pathlib import Path
config_name = "config.py"
config_locations = (
    Path.cwd(),
    Path.home(),
    Path("/etc/thisapp"),
    # Optionally Path("~thisapp").expanduser(), when an app has a "home" directory
    Path(__file__),
)
candidates = (dir / config_name for dir in config_locations)
config_paths = (path for path in candidates if path.exists())
```

We started with a list of directories showing the order in which to search for values. First, look at the configuration file found in the current working directory; then, look in the user's home directory. An `/etc/thisapp` directory (or possibly a `~thisapp` directory) can contain installation defaults. Finally, the Python library will be examined. Each candidate location for a configuration file was used to create a generator expression, assigned to the `candidates` variable. The

`config_paths` generator applies a filter so only the files that actually exist are loaded into the `ChainMap` instance.

Once we have the names of the candidate files, we can build `ChainMap` by folding each file into the map, as follows:

```
cm_config: typing.ChainMap[str, Any] = ChainMap()
for path in config_paths:
    config_layer: Dict[str, Any] = {}
    source_code = path.read_text()
    exec(source_code, globals(), config_layer)
    cm_config.maps.append(config_layer)

simulate(config.table, config.player, config.outputfile, config.samples)
```

Each file is included by creating a new, empty map that can be updated with local variables. The `exec()` function will add the file's local variables to an empty map. The new maps are appended to the `maps` attribute of the `ChainMap` object, `cm_config`.

In `ChainMap`, every name is resolved by searching through the sequence of maps and looking for the requested key and associated value. Consider loading two configuration files into `ChainMap`, giving a structure that is similar to the following example:

```
ChainMap({},
 {'betting_rule': Martingale(),
 ...
 },
 {'betting_rule': Flat(),
 ...
 })
```

Here, many of the details have been replaced with `...` to simplify the output. The chain has a sequence of three maps:

1. The first map is empty. When values are assigned to the `ChainMap` object, they go into this initial map, which will be searched first.
2. The second map is from the most local file, that is, the first file loaded into the map; they are overrides to the defaults.
3. The last map has the application defaults; they will be searched last.

The only downside is that the reference to the configuration values will be using dictionary notation, for example, `config['betting_rule']`. We can extend `ChainMap()` to implement the attribute access in addition to the dictionary item access.

Here's a subclass of `ChainMap`, which we can use if we find the `getitem()` dictionary notation too cumbersome:

```
class AttrChainMap(ChainMap):

    def __getattr__(self, name: str) -> Any:
        if name == "maps":
            return self.__dict__["maps"]
        return super().get(name, None)

    def __setattr__(self, name: str, value: Any) -> None:
        if name == "maps":
            self.__dict__["maps"] = value
            return
        self[name] = value
```

We can now say `config.table` instead of `config['table']`. This reveals an interesting restriction on our extension to `ChainMap`, that is, we can't use `maps` as an attribute. The `maps` key is a first-class attribute of the parent `ChainMap` class, and must be left untouched by this extension.

We can define mappings from keys to values using a number of different syntaxes. In the next section, we'll take a look at JSON and YAML format for defining the parameter values.

Storing the configuration in JSON or YAML files

We can store configuration values in JSON or YAML files with relative ease. The syntax is designed to be user-friendly. While we can represent a wide variety of things in YAML, we're somewhat restricted to representing a narrower variety of object classes in JSON. We can use a JSON configuration file that is similar to the following code:

```
{
    "table":{
        "dealer":"Hit17",
        "split":"NoResplitAces",
        "decks":6,
        "limit":50,
        "payout":[3,2]
    },
    "player":{
        "play":"SomeStrategy",
        "betting":"Flat",
        "rounds":100,
        "stake":50
    },
    "simulator":{
        "samples":100,
        "outputfile":"p2_c13_simulation.dat"
    }
}
```

The JSON document looks like a dictionary of dictionaries; this is precisely the same object that will be built when we load this file. We can load a single configuration file using the following code:

```
import json
config = json.load("config.json")
```

This allows us to use `config['table']['dealer']` to look up the specific class to be used for the dealer's rules. We can use `config['player']['betting']` to locate the player's particular betting strategy class name.

Unlike INI files, we can easily encode `tuple` like a sequence of values. So, the `config['table']['payout']` value will be a proper two-element sequence. It won't, strictly speaking, be `tuple`, but it will be close enough for us to use it without

having to use `ast.literal_eval()`.

Here's how we'd use this nested structure. We'll only show you the first part of the `main_nested_dict()` function:

```
def main_nested_dict(config: Dict[str, Any]) -> None:
    dealer_nm = config.get("table", {}).get("dealer", "Hit17")
    dealer_rule = {
        "Hit17": Hit17(),
        "Stand17": Stand17()
    }.get(dealer_nm, Hit17())
    split_nm = config.get("table", {}).get("split", "ReSplit")
    split_rule = {
        "ReSplit": ReSplit(),
        "NoReSplit": NoReSplit(),
        "NoReSplitAces": NoReSplitAces()
    }.get(split_nm, ReSplit())
    decks = config.get("table", {}).get("decks", 6)
    limit = config.get("table", {}).get("limit", 100)
    payout = config.get("table", {}).get("payout", (3, 2))
    table = Table(
        decks=decks, limit=limit, dealer=dealer_rule, split=split_rule, payout=payout
    )
```

This is very similar to the `main_ini()` function mentioned previously. When we compare this with the preceding version, using `configparser`, it's clear that the complexity is almost the same. The naming is slightly simpler, that is, we use `config.get('table',{}).get('decks')` instead of `config.getint('table','decks')`.

The main difference is shown in the highlighted line. JSON format provides us with properly decoded integer values and proper sequences of values. We don't need to use `eval()` or `ast.literal_eval()` to decode the tuple. The other parts, to build `Player` and configure the `Simulate` object, are similar to the `main_ini()` version.

In some cases, the nested structure of a JSON file can be confusing to edit. One way to simplify the syntax is to use a slightly different approach to organizing the data. In the next section, we'll explore a way to remove some of the complexity by using a flatter structure.

Using flattened JSON configurations

If we want to provide for default values by integrating multiple configuration files, we can't use both `ChainMap` and a nested dictionary-of-dictionaries like this. We have to either flatten out our program's parameters or look at an alternative to merging the parameters from different sources.

We can easily flatten the names by using simple `.` separators between names to reflect a top-level section and a lower-level property within the section. In that case, our JSON file might look like the following code:

```
{'player.betting': 'Flat',
  'player.play': 'SomeStrategy',
  'player.rounds': '100',
  'player.stake': '50',
  'simulator.outputfile': 'data/ch14_simulation5.dat',
  'simulator.samples': '100',
  'table.dealer': 'Hit17',
  'table.decks': '6',
  'table.limit': '50',
  'table.payout': '(3,2)',
  'table.split': 'NoResplitAces'}
```

This has the advantage of allowing us to use `ChainMap` to accumulate the configuration values from various sources. It also slightly simplifies the syntax to locate a particular parameter value. Given a list of configuration filenames, `config_names`, we might do something like this:

```
config = ChainMap(*[json.load(file) for file in config_names])
```

This builds a proper `ChainMap` from a list of configuration filenames. Here, we're loading a list of `dict` literals into `ChainMap` and the first `dict` literal will be the first one searched for by the key.

We can use a method like this to exploit `ChainMap`. We'll only show you the first part, which builds the `Table` instance:

```
def main_cm(config: Dict[str, Any]) -> None:
    dealer_nm = config.get("table.dealer", "Hit17")
    dealer_rule = {"Hit17": Hit17(), "Stand17": Stand17()}.get(dealer_nm, Hit17())
    split_nm = config.get("table.split", "ReSplit")
    split_rule = {
        "ReSplit": ReSplit(),
        "NoReSplit": NoReSplit(),
```

```
        "NoReSplitAces": NoReSplitAces()
    }.get(
        split_nm, ReSplit()
    )
    decks = int(config.get("table.decks", 6))
    limit = int(config.get("table.limit", 100))
    payout = config.get("table.payout", (3, 2))
    table = Table(
        decks=decks,
        limit=limit,
        dealer=dealer_rule,
        split=split_rule,
        payout=payout
    )
```

The other parts, to build `Player` and configure the `Simulate` object, are similar to the `main_ini()` version. However, they're omitted from this example.

When we compare this to the previous version, using `configparser`, it's clear that the complexity is almost the same. But, the naming is slightly simpler; here, we use `int(config.get('table.decks'))` instead of `config.getint('table','decks')`.

The JSON format for properties is convenient to use. The syntax, however, isn't very friendly for people to use. In the next section, we'll take a look at using YAML syntax instead of JSON syntax.

Loading a YAML configuration

As YAML syntax contains JSON syntax, the previous examples can be loaded with YAML as well as JSON. Here's a version of the nested dictionary-of-dictionaries technique from the JSON file:

```
# Complete Simulation Settings
table: !!python/object:Chapter_14.simulation_model.Table
  dealer: !!python/object:Chapter_14.simulation_model.Hit17 {}
  decks: 6
  limit: 50
  payout: !!python/tuple [3, 2]
  split: !!python/object:Chapter_14.simulation_model.NoReSplitAces {}
player: !!python/object:Chapter_14.simulation_model.Player
  betting:  !!python/object:Chapter_14.simulation_model.Flat {}
  init_stake: 50
  max_rounds: 100
  play: !!python/object:Chapter_14.simulation_model.SomeStrategy {}
  rounds: 0
  stake: 63.0
samples: 100
outputfile: data/ch14_simulation4c.dat
```

This is often easier for people to edit than pure JSON. For applications where the configuration is dominated by strings and integers, this has a number of advantages. The process to load this file is the same as the process to load the JSON file:

```
import yaml
config = yaml.load("config.yaml")
```

This has the same limitations as the nested dictionaries. We don't have an easy way to handle default values unless we flatten out the names.

When we move beyond simple strings and integers, however, we can try to leverage YAML's ability to encode class names and create instances of our customized classes. Here's a YAML file that will directly build the configuration objects that we need for our simulation:

```
# Complete Simulation Settings
table: !!python/object:__main__.Table
  dealer: !!python/object:__main__.Hit17 {}
  decks: 6
  limit: 50
  payout: !!python/tuple [3, 2]
  split: !!python/object:__main__.NoReSplitAces {}
player: !!python/object:__main__.Player
```

```
  betting:  !!python/object:__main__.Flat {}
  init_stake: 50
  max_rounds: 100
  play: !!python/object:__main__.SomeStrategy {}
  rounds: 0
  stake: 63.0
samples: 100
outputfile: data/ch14_simulation4c.dat
```

We have encoded class names and instance construction in YAML, allowing us to define the complete initialization for `Table` and `Player`. We can use this initialization file as follows:

```
import yaml

if __name__ == "__main__":

    config = yaml.load(yaml1_file)
    print(config)

    simulate(
        config["table"],
        config["player"],
        Path(config["outputfile"]),
        config["samples"]
    )
```

This shows us that a YAML configuration file can be used for human editing. YAML provides us with the same capabilities as Python but with a different syntax. For this type of example, a Python configuration script might be better than YAML.

Another format available for configuration parameters is called the properties file. We'll examine the structure and parsing of properties files, and learn how to use them in the next section.

Storing the configuration in properties files

The properties files are often used with Java programs. However, there's no reason why we can't use them with Python. They're relatively easy to parse and allow us to encode the configuration parameters in a handy, easy-to-use format. For more information about the format, you can refer to `http://en.wikipedia.org/wiki/.properties` and `https://docs.oracle.com/javase/10/docs/api/java/util/Properties.html#load(java.io.Reader)`.

Here's what a properties file might look like:

```
# Example Simulation Setup

player.betting: Flat
player.play: SomeStrategy
player.rounds: 100
player.stake: 50

table.dealer: Hit17
table.decks: 6
table.limit: 50
table.payout: (3,2)
table.split: NoResplitAces

simulator.outputfile = data/ch14_simulation5.dat
simulator.samples = 100
```

This has some advantages in terms of simplicity. The `section.property` qualified names are commonly used to structure related properties into sections. These can become long in a very complex configuration file if too many levels of nesting are used.

This format has a great deal of flexibility. The individual lines, however, can be parsed to create a mapping from property name to property value. In the next section, we'll take a look at parsing a properties file.

Parsing a properties file

There's no built-in properties parser in the Python standard library. We can download a properties file parser from the Python Package Index (`https://pypi.python.org/pypi`). However, it's not a very complex class, and it's a good exercise in advanced object-oriented programming.

We'll break the class down into the top-level API functions and the lower-level parsing functions. Here are some of the overall API methods:

```
import re

class PropertyParser:

    def read_string(self, data: str) -> Iterator[Tuple[str, str]]:
        return self._parse(data)

    def read_file(self, file: IO[str]) -> Iterator[Tuple[str, str]]:
        data = file.read()
        return self.read_string(data)

    def read(self, path: Path) -> Iterator[Tuple[str, str]]:
        with path.open("r") as file:
            return self.read_file(file)
```

The essential feature here is that it will parse a filename, a file, or a block of text. This follows the design pattern from `configparser`. A common alternative is to have fewer methods and use `isinstance()` to determine the type of the argument and to also determine what processing to perform on it.

Filenames are given as `Path` objects. While the file is generally an instance of `io.TextIOBase`, the typing module provides the `IO[str]` hint; a block of text is also a string. For this reason, many libraries use `load()` to work with files or filenames and use `loads()` to work with a simple string. Something like this would echo the design pattern of `json`:

```
def load(self, file_name_or_path: Union[TextIO, str, Path]) -> Iterator[Tuple[str, str]]
    if isinstance(file_name_or_path, io.TextIOBase):
        return self.loads(file_name_or_path.read())
    else:
        name_or_path = cast(Union[str, Path], file_name_or_path)
        with Path(name_or_path).open("r") as file:
            return self.loads(file.read())

def loads(self, data: str) -> Iterator[Tuple[str, str]]:
    return self._parse(data)
```

These methods will also handle a file, filename, or block of text. When a file is provided, it can be read and parsed. When a path or a string is provided, it's used to open a file with the given name. These extra methods give us an alternative API that might be easier to work with. The deciding factor is achieving a coherent design among the various libraries, packages, and modules. Here's the `_parse()` method:

```
key_element_pat = re.compile(r"(.*?)\s*(?<!\\)[:=\s]\s*(.*)")

def _parse(self, data: str) -> Iterator[Tuple[str, str]]:
    logical_lines = (
        line.strip() for line in re.sub(r"\\\n\s*", "", data).splitlines()
    )
    non_empty = (line for line in logical_lines if len(line) != 0)
    non_comment = (
        line
        for line in non_empty
        if not (line.startswith("#") or line.startswith("!"))
    )
    for line in non_comment:
        ke_match = self.key_element_pat.match(line)
        if ke_match:
            key, element = ke_match.group(1), ke_match.group(2)
        else:
            key, element = line, ""
        key = self._escape(key)
        element = self._escape(element)
        yield key, element
```

This method starts with three generator expressions to handle some overall features of the physical lines and logical lines within a properties file. The generator expressions separate three syntax rules. Generator expressions have the advantage of being executed lazily; this means that no intermediate results are created from these expressions until they're evaluated by the `for line in non_comment` statement.

The first expression, assigned to `logical_lines`, merges physical lines that end with `\` to create longer logical lines. The leading (and trailing) spaces are stripped away, leaving just the line content. The `r"\\\n\s*"` **Regular Expression** (**RE**) is intended to locate continuations. It matches `\` at the end of a line and all of the leading spaces from the next line.

The second expression, assigned to `non_empty`, will only iterate over lines with a nonzero length; note that blank lines will be rejected by this filter.

The third expression, `non_comment`, will only iterate over lines that do not start with `#` or `!`. Lines that start with `#` or `!` will be rejected by this filter; this eliminates

comment lines.

Because of these three generator expressions, the `for line in non_comment` loop only iterates through non-comment, non-blank, and logical lines that are properly stripped of extra spaces. The body of the loop picks apart each remaining line to separate the key and element and then apply the `self._escape()` function to expand any escape sequences.

The key-element pattern, `key_element_pat`, looks for explicit separators of non-escaped characters, such as `:`, `=`, or a space. This pattern uses the negative lookbehind assertion, that is, a RE of `(?<!\\)`, to indicate that the following RE must be non-escaped; therefore, the following pattern must *not* be preceded by `\`. The `(?<!\\)[:=\s]` subpattern matches the non-escaped `:`, `=`, or space characters. It permits a strange-looking property line such as `a\:b: value`; the property is `a:b`. The element is `value`. The `:` in the key must be escaped with a preceding `\`.

The brackets in the RE capture the property and the element associated with it. If a two-part key-element pattern can't be found, there's no separator and the line is just the property name, with an element of `""`.

The properties and elements form a sequence of two-tuples. The sequence can easily be turned into a dictionary by providing a configuration map, which is similar to other configuration representation schemes that we've seen. They can also be left as a sequence to show the original content of the file in a particular order. The final part is a small method function to transform any escape sequences in an element to their final Unicode character:

```
def _escape(self, data: str) -> str:
    d1 = re.sub(r"\\([:#!=\s])", lambda x: x.group(1), data)
    d2 = re.sub(r"\\u([0-9A-Fa-f]+)", lambda x: chr(int(x.group(1), 16)), d1)
    return d2
```

This `_escape()` method function performs two substitution passes. The first pass replaces the escaped punctuation marks with their plain-text versions: `\:`, `\#`, `\!`, `\=`, and `\` all have `\` removed. For the Unicode escapes, the string of digits is used to create a proper Unicode character that replaces the `\uxxxx` sequence. The hex digits are turned into an integer, which is turned into a character for the replacement.

The two substitutions can be combined into a single operation to save creating

an intermediate string that will only get discarded. This will improve the performance; it should look like the following code:

```
d2 = re.sub(
    r"\\([:#!=\s])|\\u([0-9A-Fa-f]+)",
    lambda x: x.group(1) if x.group(1) else chr(int(x.group(2), 16)),
    data,
)
```

The benefit of better performance might be outweighed by the complexity of the RE and the replacement function.

Once we have parsed the properties values, we need to use them in an application. In the next section, we'll examine ways of using a properties file.

Using a properties file

We have two choices for how we use a properties file. We could follow the design pattern of `configparser` and parse multiple files to create a single mapping from the union of the various values. Alternatively, we could follow the `ChainMap` pattern and create a sequence of property mappings for each configuration file.

The `ChainMap` processing is reasonably simple and provides us with all the required features:

```
pp = PropertyParser()

candidate_list = [prop_file]
config = ChainMap(
    *[dict(pp.read_file(file))
      for file in reversed(candidate_list)
      ]
)
```

We've taken the list in reverse order: the most specific settings will be first in the internal list, while the most general settings will be the last. Once `ChainMap` has been loaded, we can use the properties to initialize and build our `Player`, `Table`, and `Simulate` instances.

This seems simpler than updating a single mapping from several sources. Additionally, this follows the pattern used to process JSON or YAML configuration files.

We can use a method like this to exploit `ChainMap`. This is very similar to the `main_cm()` function mentioned previously. We'll only show you the first part, which builds the `Table` instance:

```
import ast

def main_cm_prop(config):
    dealer_nm = config.get("table.dealer", "Hit17")
    dealer_rule = {"Hit17": Hit17(), "Stand17": Stand17()}.get(dealer_nm, Hit17())
    split_nm = config.get("table.split", "ReSplit")
    split_rule = {
        "ReSplit": ReSplit(), "NoReSplit": NoReSplit(), "NoReSplitAces": NoReSplitAces()
    }.get(
        split_nm, ReSplit()
    )
    decks = int(config.get("table.decks", 6))
    limit = int(config.get("table.limit", 100))
```

```
    payout = ast.literal_eval(config.get("table.payout", "(3,2)"))
    table = Table(
        decks=decks, limit=limit, dealer=dealer_rule, split=split_rule, payout=payout
    )
```

The difference between this version and the `main_cm()` function is the handling of the payout tuple. In the previous version, JSON (and YAML) could parse the tuple. When using the properties files, all values are simple strings. We must use `eval()` or `ast.literal_eval()` to evaluate the given value. The other portions of this `main_cm_str()` function are identical to `main_cm()`.

Using the properties file format is one way to persist configuration data. In the next section, we'll look at using XML syntax to represent configuration parameters.

Using XML files – PLIST and others

As we noted in Chapter 10, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML*, Python's `xml` package includes numerous modules that parse the XML files. Because of the wide adoption of the XML files, it often becomes necessary to convert between XML documents and Python objects. Unlike JSON or YAML, the mapping from XML is not simple.

One common way to represent the configuration data in XML is the PLIST file. For more information on the PLIST format, you can refer to https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/PropertyLists/Introduction/Introduction.html.

Macintosh users with XCode installed can perform `man plist` to see extensive documentation on the XML-based format. The advantage of the PLIST format is that it uses a few, very general tags. This makes it easy to create PLIST files and parse them. Here's the sample PLIST file with our configuration parameters:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/Property
<plist version="1.0">
<dict>
  <key>player</key>
  <dict>
    <key>betting</key>
    <string>Flat</string>
    <key>play</key>
    <string>SomeStrategy</string>
    <key>rounds</key>
    <integer>100</integer>
    <key>stake</key>
    <integer>50</integer>
  </dict>
  <key>simulator</key>
  <dict>
    <key>outputfile</key>
    <string>ch14_simulation6a.dat</string>
    <key>samples</key>
    <integer>100</integer>
  </dict>
  <key>table</key>
  <dict>
    <key>dealer</key>
    <string>Hit17</string>
    <key>decks</key>
    <integer>6</integer>
    <key>limit</key>
    <integer>50</integer>
    <key>payout</key>
```

```
    <array>
      <integer>3</integer>
      <integer>2</integer>
    </array>
    <key>split</key>
    <string>NoResplitAces</string>
  </dict>
</dict>
</plist>
```

Here, we're showing you the nested dictionary-of-dictionary structure in this example. There are a number of Python-compatible types encoded with XML tags:

Python type	Plist tag
`str`	`<string>`
`float`	`<real>`
`int`	`<integer>`
`datetime`	`<date>`
`boolean`	`<true/> or <false/>`
`bytes`	`<data>`
`list`	`<array>`
`dict`	`<dict>`

As shown in the preceding example, the dict `<key>` values are strings. This makes

the PLIST file a very pleasant encoding of our parameters for our simulation application. We can load a PLIST-compliant XML file with relative ease:

```
import plistlib
print(plistlib.load(plist_file))
```

This will reconstruct our configuration parameter from the XML serialization. We can then use this nested dictionary-of-dictionaries structure with the `main_nested_dict()` function shown in the preceding section on JSON configuration files.

Using a single module function to parse the file makes the PLIST format very appealing. The lack of support for any customized Python class definitions makes this equivalent to JSON or a properties file.

In addition to the standardized PLIST schema, we can define our own customized schema. In the next section, we'll look at creating XML that's unique to our problem domain.

Customized XML configuration files

For a more complex XML configuration file, you can refer to http://wiki.metawerx.net/wiki/Web.xml. These files contain a mixture of special-purpose tags and general-purpose tags. These documents can be challenging to parse. There are two general approaches:

- Write a document processing class that uses XPath queries to locate the tags in the XML document structure. In this case, we'll create a class with properties (or methods) to locate the requested information in the XML document.
- Unwind the XML document into a Python data structure. This is the approach followed by the `plist` module, mentioned previously. This will convert the XML text values into native Python objects.

Based on examples of the `web.xml` files, we'll design our own customized XML document to configure our simulation application:

```
<?xml version="1.0" encoding="UTF-8"?>
<simulation>
    <table>
        <dealer>Hit17</dealer>
        <split>NoResplitAces</split>
        <decks>6</decks>
        <limit>50</limit>
        <payout>(3,2)</payout>
    </table>
    <player>
        <betting>Flat</betting>
        <play>SomeStrategy</play>
        <rounds>100</rounds>
        <stake>50</stake>
    </player>
    <simulator>
        <outputfile>data/ch14_simulation6b.dat</outputfile>
        <samples>100</samples>
    </simulator>
</simulation>
```

This is a specialized XML file. We didn't provide a DTD or an XSD, so there's no formal way to validate the XML against a schema. However, this file is small, easily debugged, and parallels other example initialization files. Here's a `Configuration` class that can use XPath queries to retrieve information from this file:

```
import xml.etree.ElementTree as XML

class Configuration:

    def read_file(self, file):
        self.config = XML.parse(file)

    def read(self, filename):
        self.config = XML.parse(filename)

    def read_string(self, text):
        self.config = XML.fromstring(text)

    def get(self, qual_name, default):
        section, _, item = qual_name.partition(".")
        query = "./{0}/{1}".format(section, item)
        node = self.config.find(query)
        if node is None:
            return default
        return node.text

    def __getitem__(self, section):
        query = "./{0}".format(section)
        parent = self.config.find(query)
        return dict((item.tag, item.text) for item in parent)
```

We've implemented three methods to load the XML document: `read()`, `read_file()`, and `read_string()`. Each of these simply delegates itself to an existing method function of the `xml.etree.ElementTree` class. This parallels the `configparser` API. We could use `load()` and `loads()` method names too, as they would delegate themselves to `parse()` and `fromstring()`, respectively.

For access to the configuration data, we implemented two methods: `get()` and `__getitem__()`. These methods build XPath queries to locate the section and item within the XML structure. The `get()` method allows us to use code like this: `stake = int(config.get('player.stake', 50))`. The `__getitem__()` method allows us to use code like this: `stake = config['player']['stake']`.

The parsing is a trifle more complex than a PLIST file. However, the XML document is much simpler than an equivalent PLIST document.

We can use the `main_cm_prop()` function, mentioned in the previous section on the properties files, to process this configuration.

Summary

In this chapter, we explored a number of ways to represent the configuration parameters. Most of these are based on more general serialization techniques that we saw in `Chapter 10`, *Serializing and Saving – JSON, YAML, Pickle, CSV, and XML*. The `configparser` module provides an additional format that's comfortable for some users.

The key feature of a configuration file is that the content can be easily edited by a human. For this reason, pickle files aren't recommended as a good representation.

Design considerations and trade-offs

Configuration files can simplify running application programs or starting servers. This can put all the relevant parameters in one easy-to-read and easy-to-modify file. We can put these files under the configuration control, track change history, and generally use them to improve the software's quality.

We have several alternative formats for these files, all of which are reasonably human-friendly to edit. They vary in how easy they are to parse and any limitations on the Python data that can be encoded:

- **INI files**: These files are easy to parse and are limited to strings and numbers.
- **Python code** (**PY files**): We can use the main script for the configuration; in this case, there will be no additional parsing and no limitations. We can also use `exec()` to process a separate file; this makes it trivial to parse and, again, there are no limitations.
- **JSON** or **YAML files**: These files are easy to parse. They support strings, numbers, dicts, and lists. YAML can encode Python, but then why not just use Python?
- **Properties files**: These files require a special parser. They are limited to strings.
- **XML files**:
  - **PLIST files**: These files are easy to parse. They support strings, numbers, dicts, and lists.
  - **Customized XML**: These files require a special parser. They are limited to strings, but a mapping to a Python object allows a variety of conversions to be performed by the class.

Coexistence with other applications or servers will often determine a preferred format for the configuration files. If we have other applications that use PLIST or INI files, then our Python applications should make choices that are more comfortable for users to work with.

Viewed from the breadth of objects that can be represented, we have four broad categories of configuration files:

- **Simple files with only strings**: Custom XML and properties files.
- **Simple files with Python literals**: INI files.
- **More complex files with Python literals, lists, and dicts**: JSON, YAML, PLIST, and XML.
- **Anything that is Python**: We can use YAML for this, but it seems silly when Python has a clearer syntax than YAML. Providing configuration values through Python class definitions is very simple and leads to a pleasant hierarchy of default and override values.

Creating a shared configuration

When we look at module design considerations in `Chapter 19`, *Module and Package Design*, we'll see how a module conforms to the **Singleton** design pattern. This means that we can import a module only once, and the single instance is shared.

Because of this, it's often necessary to define a configuration in a distinct module and import it. This allows separate modules to share a common configuration. Each module will import the shared configuration module; the configuration module will locate the configuration file(s) and create the actual configuration objects.

Schema evolution

The configuration file is part of the public-facing API. As application designers, we have to address the problem of schema evolution. If we change a class definition, how will we change the configuration?

Because configuration files often have useful defaults, they are often very flexible. In principle, the content is entirely optional.

As a piece of software undergoes major version changes – changes that alter the APIs or the database schema – the configuration files too might undergo major changes. The configuration file's version number may have to be included in order to disambiguate legacy configuration parameters from current release parameters.

For minor version changes, the configuration files, such as database, input and output files, and APIs, should remain compatible. Any configuration parameter handling should have appropriate alternatives and defaults to cope with minor version changes.

A configuration file is a first-class input to an application. It's not an afterthought or a workaround. It must be as carefully designed as the other inputs and outputs. When we look at larger application architecture design in `Chapter 16`, *The Logging and Warning Modules*, and `Chapter 18`, *Coping with the Command Line*, we'll expand on the basics of parsing a configuration file.

Looking forward

In the following chapters, we'll explore larger-scale design considerations. `Chapter 15`, *Design Principles and Patterns* will address some general principles that can help structure the class definitions of an object-oriented program. `Chapter 16`, *The Logging and Warning Modules* will look at using the `logging` and `warnings` modules to create audit information as well as to debug. We'll look at designing for testability and how we use `unittest` and `doctest` in `Chapter 17`, *Designing for Testability*. `Chapter 18`, *Coping with the Command Line* will look at using the `argparse` module to parse options and arguments. We'll take this a step further and use the **Command** design pattern to create program components that can be combined and expanded without resorting to writing shell scripts. In `Chapter 19`, *Module and Package Design,* we'll look at module and package design. In `Chapter 20`, *Quality and Documentation,* we'll look at how we can document our design to ensure that our software is correct and is properly implemented.

Section 3: Object-Oriented Testing and Debugging

A finished application includes automated unit testing, as well as support for debugging via the Python logging system. Beyond that, documentation of the application's structure, support, and operation is also essential. This section looks at logging, testing, and documenting your code.

The following chapters will be covered in this section:

- `Chapter 15`, *Design Principles and Patterns*
- `Chapter 16`, *The Logging and Warning Modules*
- `Chapter 17`, *Designing for Testability*
- `Chapter 18`, *Coping with the Command Line*
- `Chapter 19`, *Module and Package Design*
- `Chapter 20`, *Quality and Documentation*

Design Principles and Patterns

There are a number of considerations for object-oriented design. In this chapter, we'll take a step back from the details of the Python language to look at a few general principles. These principles provide fundamental guidance on how to design stateful objects. We'll look at concrete applications of the principles in Python.

The general approach we'll follow is defined by the **SOLID design principles**, which are as follows:

- **S**ingle Responsibility
- **O**pen/Closed
- **L**iskov Substitution
- **I**nterface Segregation
- **D**ependency Inversion

While the principles have a clever mnemonic, we won't cover them in this order. The **Interface Segregation Principle** (**ISP**) seems to be the most helpful for decomposing a complex problem into individual class definitions. Most of the remaining principles help to refine the features of class definitions. The **Single Responsibility Principle** (**SRP**) seems a little less focused than the others, making it more useful as a summary than a starting point. For more information, see `http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod` for the original concepts. The linked site also includes a number of additional concepts. Our purpose in this book is to provide some Pythonic context to these principles. In this chapter, we will cover the following topics:

- The SOLID design principles
- SOLID principle design test
- Building features through inheritance or composition
- Parallels between Python and libstdc++

Technical requirements

The code files for this chapter are available at `https://git.io/fj2UX.`

The SOLID design principles

One goal for the **SOLID** design principles is to limit the effects of change or extension on a design. Making a change to established software is a bit like casting a pebble into the sea: there will be an initial splash, followed by ripples of change spreading outward. When trying to fix or extend badly designed software, the initial splash radius covers everything; the ripples are large and lead to numerous problems. In well-designed software, the splash radius is tiny.

As a concrete example, consider a class to represent dominoes. Each tile has 2 numbers, from 0 to 6, leading to 28 distinct tiles. The class design looks more or less like a two-tuple. The overall collection of 28 tiles can be generated with a nested pair of `for` statements, or a generator expression with two `for` clauses.

In some games, however, the tiles could have an upper limit of 9, 12, or even 15. Having different upper limits leads to a change to the class that represents the overall collection of dominoes. In Python, this change may be as tiny as adding a default parameter, `limit=6`, to an object constructor. In poorly-designed software, the number `6` appears in more than one place in the class definition, and the change has a large splash radius.

While playing, some dominoes with double numbers, especially the double-six, can have special roles. In some games, double-numbers tiles are called *spinners* and have a dramatic impact on the state of play. In other games, the double-number tiles are merely used to determine who plays first, and have little overall impact. These changes in role can lead to changes in the design for a class definition. A **SOLID** design will isolate changes to one portion of the application, limiting the ripples from disturbing unrelated portions of the software.

In many games, the sum of the two numbers is the value of the tile; the winner's points are based on the sum of the values of the unplayed tiles in the other player's hands. This means that tiles with higher point values should often be played first, and tiles with lower point values represent less risk. This suggests a variety of policies for sorting tiles into order, leading to design variations.

All of these variations in the rules suggest we need to be flexible in our class design. If we focus on the rules for one specific game, then our class definition cannot easily be reused or modified for other games. To maximize the value of a class definition means providing a class general enough to solve a number of closely related problems. We'll start with a class definition containing a number of design flaws. The poor design is as follows:

```
import random
from typing import Tuple, List, Iterator

class DominoBoneYard:

    def __init__(self, limit: int = 6) -> None:
        self._dominoes = [
            (x, y) for x in range(limit + 1)
            for y in range(x + 1)
        ]
        random.shuffle(self._dominoes)

    def double(self, domino: Tuple[int, int]) -> bool:
        x, y = domino
        return x == y

    def score(self, domino: Tuple[int, int]) -> int:
        return domino[0] + domino[1]

    def hand_iter(self, players: int = 4) -> Iterator[List[Tuple[int, int]]]:
        for p in range(players):
            yield self._dominoes[p * 7:p * 7 + 7]

    def can_play_first(self, hand: List[Tuple[int, int]]) -> bool:
 for d in hand:
 if self.double(d) and d[0] == 6:
 return True
 return False

    def score_hand(self, hand: List[Tuple[int, int]]) -> int:
        return sum(d[0]+d[1] for d in hand)

    def rank_hand(self, hand: List[Tuple[int, int]]) -> None:
        hand.sort(key=self.score, reverse=True)

    def doubles_indices(self, hand: List[Tuple[int, int]]) -> List[int]:
        return [i for i in range(len(hand)) if self.double(hand[i])]
```

While this class can be used for some common games, it has many problems. If we want to extend or modify this definition for other games, almost any change seems to create a large splash with extensive ripples.

We'll call out a particularly difficult method, the `can_play_first()` method. In a double-6 game with 4 players, it's common for all 28 dominoes to be dealt. One of the 4 hands will have the double-6; that player goes first. In a 2-player variation, however, only 14 dominoes will be dealt, and there's a 50% chance

that neither player has the double-6. To cover that common case, the rule is often stated as *highest double plays first*. This class doesn't *easily* find any double other than double-6.

This class should have been decomposed into a number of smaller classes. We'll look at each of the **SOLID** design principles in the following sections to see how they guide us toward better design. We want to build object models to better support solving a variety of closely related problems.

The Interface Segregation Principle

One definition of the **Interface Segregation Principle** (**ISP**) is that c*lients should not be forced to depend on methods that they do not use*. The idea is to provide the smallest number of methods in a class. This leads to focused definitions, often separating a design into multiple classes to isolate the impact of any changes.

This principle seems to have the most dramatic impact on a design because it decomposes the model into a number of classes, each with a focused interface. The other four principles seem to follow from this beginning by providing improvements after the initial decomposition.

The type hints embedded in the class definition shown earlier suggest there are at least three different interfaces involved in the class. We can see type hints for the following:

- The overall collection of dominoes, `List[Tuple[int, int]]`, used to deal hands.
- Each individual domino, defined by a type hint in the form of `Tuple[int, int]`.
- A hand of dominoes, also defined by the type hint in the form of `List[Tuple[int, int]]`. This is, perhaps, an ambiguity that is exposed by having similar types with different purposes.
- The `doubles_indices()` query about specific dominoes within a hand, where the result is `List[int]`. This may not be different enough to merit yet another class definition.

If we decompose the initial class based on these separate interfaces, we'll have a number of classes that can then evolve independently. Some classes can be reused widely in a variety of games; other classes will have to have game-specific extensions created. The revised pair of classes is shown in the following code:

```
class Domino(NamedTuple):
    v1: int
    v2: int

    def double(self) -> bool:
        return self.v1 == self.v2
```

```
    def score(self) -> int:
        return self.v1 + self.v2


class Hand(list):

    def __init__(self, *args: Domino) -> None:
        super().__init__(cast(Tuple[Any], args))

    def score(self) -> int:
        return sum(d.score() for d in self)

    def rank(self) -> None:
        self.sort(key=lambda d: d.score(), reverse=True)

    def doubles_indices(self) -> List[int]:
        return [i for i in range(len(self)) if self[i].double()]
```

The Domino class preserves the essential `Tuple[int, int]` structure, but provides a sensible name for the class and names for the two values shown on the tile. A consequence of using a `NamedTuple` is a more useful `repr()` value when we print objects.

The `__init__()` method of the `Hand` class doesn't really do much useful work. The `cast()` function applied to a type, `Type[Any]`, and the `args` object does nothing at runtime. The `cast()` is a hint to `mypy` that the values of `args` should be considered as having the `Tuple[Any]` type, instead of the more restrictive `Domino` type. Without this, we get a misleading error about the `list.__init__()` method expecting objects of the `Any` type.

The score of a `Hand` instance depends on scores of the various `Domino` objects in the `Hand` collection. Compare this with the `score_hand()` and `score()` functions shown previously. The poor design repeats important algorithmic details in two places. A small change to one of these places must also be made to another place, leading to a wider splash from a change.

The `double_indices()` function is rather complex because it works with index positions of dominoes rather than the domino objects themselves. Specifically, the use of `for i in range(len(self))` means the value of the `i` variable will be an integer, and `self[i]` will be the `Domino` object with the index value equal to the value of the `i` variable. This function provides the indices of dominoes for which the `double()` method is `True`.

To continue this example, the overall collection of dominoes is shown in the following code:

```
class DominoBoneYard2:

    def __init__(self, limit: int = 6) -> None:
        self._dominoes = [Domino(x, y) for x in range(limit + 1) for y in range(x + 1)]
        random.shuffle(self._dominoes)

    def hand_iter(self, players: int = 4) -> Iterator[Hand]:
        for p in range(players):
            yield Hand(self._dominoes[p * 7:p * 7 + 7])
```

This creates individual `Domino` instances when the initial set of dominoes is created. Then, it creates individual `Hand` instances when dealing dominoes to players.

Because the interfaces have been minimized, we can consider changing the way dominoes are dealt without breaking the essential definition of each tile or a hand of tiles. Specifically, the design as shown doesn't work well with the tiles not dealt to the players. In a 2-player game, for example, there will be 14 unused tiles. In some games, these are simply ignored. In other games, players are forced to choose from this pool. Adding this feature to the original class runs the risk of disturbing other interfaces, unrelated to the mechanics of dealing. Adding a feature to the `DominoBoneyard2` class doesn't introduce the risk of breaking the behavior of `Domino` or `Hand` objects.

We can, for example, make the following code change:

```
class DominoBoneYard3(DominoBoneYard2):

    def hand_iter(self, players: int = 4) -> Iterator[Hand3]:
        for p in range(players):
            hand, self._dominoes = Hand3(self._dominoes[:7]), self._dominoes[7:]
            yield hand
```

This would preserve any undealt dominoes in the `self._dominoes` sequence. A `draw()` method could consume dominoes one at a time after the initial deal. This change does not involve changes to any other class definitions; this isolation reduces the risk of introducing astonishing or confusing problems in other classes.

The Liskov Substitution Principle

The **Liskov Substitution Principle** (**LSP**) is named after computer scientist Barbara Liskov, inventor of the CLU language. This language emphasizes the concept of a *cluster* containing a description of the representation of an object and the implementations of all the operations on that object. For more information on this early object-oriented programming language, see `http://www.pmg.lcs.mit.edu/CLU.html`.

The LSP is often summarized as *subtypes must be substitutable for their base types*. This advice tends to direct us toward creating polymorphic type hierarchies. If, for example, we wish to add features to the `Hand` class, we should make sure any subclass of `Hand` can be used as a direct replacement for `Hand`.

In Python, a subclass that extends a superclass by adding new methods is an ideal design. This subclass extension demonstrates the LSP directly.

When a subclass method has a different implementation but the same type hint signature as a superclass, this also demonstrates elegant Liskov substitutability. The examples shown previously include `DominoBoneYard2` and `DominoBoneYard3`. Both of these classes have the same methods with the same type hints and parameters. The implementations are different. The subclass can substitute for the parent class.

In some cases, we'd like to have a subclass that uses additional parameters or has a slightly different type signature. A design where a subclass method doesn't match the superclass method is often less than ideal, and an alternative design should be considered. In many cases, this should be done by *wrapping* the superclass instead of extending it.

Wrapping a class to add features is a way to create a new entity without creating Liskov Substitution problems. Here is an example:

```
class FancyDealer4:

    def __init__(self):
        self.boneyard = DominoBoneYard3()
```

```
    def hand_iter(self, players: int, tiles: int) -> Iterator[Hand3]:
        if players * tiles > len(self.boneyard._dominoes):
            raise ValueError(f"Can't deal players={players} tiles={tiles}")
        for p in range(players):
            hand = Hand3(self.boneyard._dominoes[:tiles])
            self.boneyard._dominoes = self.boneyard._dominoes[tiles:]
            yield hand
```

The `FancyDealer4` class definition is not a subclass of the previous `DominoBoneYard2` or `DominoBoneYard3` classes. This wrapper defines a distinct signature for the `hand_iter()` method: this is an additional parameter, and there are no default values. Each instance of `FancyDealer4` wraps a `DominoBoneYard3` instance; this object is used to manage the details of the available tiles.

Wrapping a class makes an explicit claim that the LSP is not a feature of the class design. The choice between writing a wrapper or creating a subclass is often informed by the LSP.

Python's use of default values and keyword parameters provides a tremendous amount of flexibility. In many cases, we can consider rewriting a superclass to provide suitable defaults. This is often a way to avoid creating more subclasses or more wrapper classes. In some languages, the compiler rules for inheritance require considerable cleverness to get to a class hierarchy where a subclass can be used in place of the superclass. In Python, cleverness is rarely required; instead, we can often add optional parameters.

The Open/Closed Principle

The **Open/Closed Principle** (**OCP**) suggests two complementary objectives. On the one hand, a class should be open to extension. On the other hand, it should also be closed to modification. We want to design our classes to support extension via wrapping or subclasses. As a general habit, we'd like to avoid modifying classes. When new features are needed, a sound approach is to extend classes to add the features.

When we want to introduce changes or new features, the ideal path is via extension of existing classes. This leaves all of the legacy features in place, leaving the original tests in place to confirm no previous feature was broken by adding a new feature.

When keeping a class open to extension, there are two kinds of design changes or adaptations that arise:

- A subclass needs to be added where the method signatures match the parent class. The subclass may have additional methods, but it will contain all of the parent class features and can be used in place of the parent class. This design also follows the LSP.
- A wrapper class needs to be added to provide additional features that are not compatible with another class hierarchy. A wrapper class steps outside direct Liskov Substitution because the new features of the wrapper will not be directly compatible with the other classes.

In either case, the original class definitions remain unmodified as the design evolves. The new features are either extensions that satisfy the LSP, or wrappers that create a new class from an old class definition. This kind of design is a consequence of keeping the classes open to extension.

Our example classes, `DominoBoneYard2` and `DominoBoneYard3`, shown previously, both suffer from a profound failure to follow the OCP. In both of these classes, the number of tiles in a hand is fixed at seven. This literal value makes the class difficult to extend. We were forced to create the `FancyDealer4` class to work around this design flaw.

A better design of the `DominoBoneYard2` class would lead to easier extension to all of the classes in this hierarchy. A small change that works very nicely in Python is to make the constant value into a class-level attribute. This change is shown in the following code sample:

```
class DominoBoneYard2b:

    hand_size: int = 7

    def __init__(self, limit: int = 6) -> None:
        self._dominoes = [Domino(x, y) for x in range(limit + 1) for y in range(x + 1)]
        random.shuffle(self._dominoes)

    def hand_iter(self, players: int = 4) -> Iterator[Hand3]:
        for p in range(players):
            hand = Hand3(self._dominoes[:self.hand_size])
            self._dominoes = self._dominoes[self.hand_size:]
            yield hand
```

The `DominoBoneYard2b` class introduces a class-level variable to make the size of each hand into a parameter. This makes the class more open to extension: a subclass can make changes without the need to modify any further programming. This isn't always the best kind of rework, but it has the advantage of being a very small change. The Python language facilitates these kinds of changes. The `self.hand_size` reference can either be a property of the instance, or a property of the class as a whole.

There are other places we can open this class to extension. We'll look at some of them as part of the Dependency Inversion Principle.

The Dependency Inversion Principle

The **Dependency Inversion Principle** (**DIP**) has an unfortunate name; the word *inversion* seems to imply there's some kind of obvious dependency and we should *invert* the obvious dependency rules. Practically, the principle is described as having class dependencies based on the most abstract superclass possible, not on a specific, concrete implementation class.

In languages with formal type declarations, for example, Java or C++, this advice to refer to abstract superclasses can be helpful to avoid complex recompiles for small changes. These languages also need fairly complex dependency injection frameworks to be sure that classes can be altered via runtime configuration changes. In Python, the runtime flexibility means the advice changes somewhat.

Because Python uses duck typing, there isn't always a single, abstract superclass available to summarize a variety of alternative implementations. We may, for example, define a function parameter to be `Iterable`, telling `mypy` to permit any object that follows the `Iterable` protocol: this will include iterators as well as collections.

In Python, the DIP leads us to two techniques:

- Type hints should be as abstract as possible. In many cases, it will name a relevant protocol used by a method or a function.
- Concrete type names should be parameterized.

In our preceding examples, the various `DominoBoneYard` class definitions all suffer from a dependency problem: they all refer to concrete class names when creating the initial pool of `Domino` objects and when creating the `Hand` objects. We are not free to replace these classes as needed, but need to create subclasses to replace a reference.

A more flexible definition of the class is shown in the following example:

```
class DominoBoneYard3c:
```

```
    domino_class: Type[Domino] = Domino

    hand_class: Type[Hand] = Hand3

    hand_size: int = 7

    def __init__(self, limit: int = 6) -> None:
        self._dominoes = [
            self.domino_class(x, y) for x in range(limit + 1) for y in range(x + 1)
        ]
        random.shuffle(self._dominoes)

    def hand_iter(self, players: int = 4) -> Iterator[Hand]:
        for p in range(players):
            hand = self.hand_class(
                self._dominoes[:self.hand_size])
            self._dominoes = self._dominoes[self.hand_size:]
            yield hand
```

This example shows how the dependencies can be defined in a central place, as attributes of the class definition. This refactors the dependencies from deep within several methods to a much more visible position. We've provided a complete type hint in order to help spot potential misuse of the type expectations. For the `Domino` class, we don't have any alternatives, and the hint, `Type[Domino]`, does seem redundant. For the `Hand3` class, however, we've provided the hint of `Type[Hand]` to show the most abstract class that will be usable here.

Because these values are variables, it becomes very easy to perform dependency injection and supply configuration information at runtime. We can use code along the lines of `DominoBoneYard3c.hand_class = Hand4` to change the class used to build a hand. Generally, this should be done before any instances are created. The class identification can be taken from a configuration file and used to tailor the details of the application's operation.

We can imagine a top-level program that includes the following:

```
configuration = get_configuration()
DominoBoneYard3c.hand_class = configuration['hand_class']
DominoBoneYard3c.domino_class = configuration['domino_class']
```

Once the class definitions have the proper dependencies injected into them, the application can then work with those configured class definitions. For more ideas on providing the configuration, see `Chapter 14`, *Configuration Files and Persistence*. It's important to note that the type hints are not used to check runtime configuration values. The type hints are only used to confirm that the source code appears to be consistent in its use of objects and types.

The Single Responsibility Principle

The **Single Responsibility Principle** (**SRP**) can be the most difficult principle to understand. The general statement that "*a class should have one, and only one, reason to change*" shifts the definition of responsibility to the understanding of change in an object-oriented design. There are several reasons for change; the most salient reason for changing a class is to add a new feature. As noted earlier in the *Open/Closed principle* section, a feature should be added as an extension rather than a modification.

In many cases, the **Interface Segregation Principle** (**ISP**) seems to provide more concrete guidance for class design than the SRP. The SRP seems to be a summary of how a class looks when we've followed the other principles.

When we review the classes defined in the preceding examples, there are some potential changes that this principle implies. In particular, the various `DominoBoneYard` class definitions provide the features listed here:

- Build the collection of `Domino` instances.
- Deal the initial hands to the players. Often this is four hands of seven dominoes, but this rule varies from game to game. This may exhaust the collection of dominoes, or it may leave some dominoes undealt.
- When there is a collection of undealt dominoes, manage the collection by allowing players to draw, to supplement their hands.

We can claim this is a single responsibility: dealing dominoes to players. There are two different ways players get dominoes (the initial deal and drawing later in the game) and both mechanisms are part of the responsibilities of a single class. This is a fairly high level of abstraction, looking at the pool of dominoes as a single thing.

We can also claim there are two responsibilities here. We can argue that creating the initial collection of `Domino` objects is a different responsibility from dealing `Domino` objects to players. We can counter-argue that adding and removing dominoes is the single responsibility of maintaining the collection contents. This is a fairly low level of abstraction.

The general guiding principles often lead to situations where expert judgment is required to make the final decision. There is no simple rule for distinguishing the level of abstraction appropriate for a design.

These principles must be used to guide the design process. They aren't immutable laws. The final decisions depend on a number of other factors, like the overall complexity of the application and the sweep of anticipated programming changes. It helps to consider the **SOLID** design principles as conversation starters. When reviewing a design, the splash radius around a change and the consequences of the change need to be evaluated and these principles provide a few dimensions for evaluating the quality of the design.

Let's take a look at the SOLID principle design test in the next section.

A SOLID principle design test

We generally think of testing as something applied to the final code (in `Chapter 17`, *Design for Testability*, we'll look at automated testing in detail). However, we can also apply a test to a SOLID design. The test is to replace a given class with an equivalent class to provide an alternative algorithm to accomplish the same purpose. If we've done our design job well, then a change to one class should have a minimal splash with few ripples.

As a concrete example, consider the `Domino` class shown earlier in this chapter, under the *Interface Segregation Principle* section. We used a `NamedTuple` to represent the pair of numbers. Some alternatives are possible:

- Use a `frozenset` to retain one or two distinct values. If there's one value in the set, the tile is actually a double, or spinner.
- Use a `Counter` to retain the counts for the values. If there's a single value, with a count of two, the tile is a double. Otherwise, there will be two values with counts of one each.

Do these kinds of change to the `Domino` class have any effect on other classes in the design? If not, then the design is nicely encapsulated. If these kinds of changes do break the design, then the design effort should continue to rearrange the class definitions in a way where the impact of a change is minimized.

In the next section, we'll build features through inheritance and composition.

Building features through inheritance and composition

As noted earlier in the *Liskov Substitution Principle* section, there are two general ways to add features to a class definition:

- Inheritance to extend a class by creating a subclass
- Composition of a new class from one or more other classes

In general, these choices are always present. Every object-oriented design decision involves choosing between inheritance and composition.

To make the decision more nuanced, Python allows multiple inheritance. While combining multiple mixing classes is partially a kind of inheritance, it is more fundamentally an exercise in composition.

The LSP can lead to avoiding inheritance in favor of composition. The general suggestion is to reserve inheritance for those situations where the child class can fully replace the parent class. When features are changed in some way to create a child that is not a direct replacement for the parent, then composition may be more appropriate.

Consider the consequences of adding some features to the `Hand` class shown earlier. Here are two examples:

- The `Hand3` subclass extended the `Hand` class by introducing an additional method. This extension is compatible with the superclass, and `Hand3` can be used as a substitute for `Hand`. This seems to be a sensible extension via inheritance.
- The `FancyDealer4` class introduced a new class composed of a new method making use of the `DominoBoneYard3` class. This class introduced a profound change to the `hand_iter()` method; the change was not trivially compatible with the superclass.

There are yet more composition techniques available in Python. In `Chapter 9`,

Decorators and Mixins – Cross-cutting Aspects, we addressed two additional composition techniques. We'll look at some other patterns of class composition in the next section.

Advanced composition patterns

One of the classic books of design patterns, *Design Patterns: Elements of Reusable Object-Oriented Software*, identified a number of common patterns of object composition. Some of these patterns are more relevant to C++ or Java programming, and less relevant to Python programming. For example, the **Singleton** pattern is a first-class aspect of a Python module and a Python class definition; the complexities of Java static variables aren't necessary to implement this pattern.

A better source for Python design patterns is available at `https://python-patterns.guide`. The pattern descriptions on this Python Patterns website are focused on Python specifically. It's important to recognize that some of the complexity in object-oriented design pattern literature stems from creating elegant ways to create a runtime behavior in the presence of very strict compile-time checking. Python doesn't suffer from the same kinds of type management issues, making Python programming simpler.

A central concept in Python is **duck typing**. The concept is based on the following quote:

"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

In Python, an object is usable when the methods and attributes fit a needed protocol. The actual base type doesn't matter; the available methods are what defines a class's suitability in a particular context.

We can, for example, define two similar-looking classes as shown in the following code. The first uses the `typing.NamedTuple` as a base class:

```
from typing import NamedTuple
from dataclasses import dataclass

class Domino_1(NamedTuple):
    v1: int
    v2: int

    @property
    def double(self):
        return self.v1 == self.v2
```

This alternative version uses a `@dataclass` decorator to create a frozen object, similar to a tuple:

```
from dataclasses import dataclass

@dataclass(frozen=True, eq=True, order=True)
class Domino_2:
    v1: int
    v2: int

    @property
    def double(self):
        return self.v1 == self.v2
```

These two classes have nearly identical behavior. The only class they have in common is the superclass for all objects, the `object` class. Yet, these two classes are functionally interchangeable, and can be freely substituted for each other.

This ability to have equivalent types *without* a common superclass permits flexibility, but can also lead to a difficulty when trying to check types with **mypy**. In some cases, we may find the need to define an abstract superclass purely for the purposes of providing assurance that several distinct implementations all provide common features. In other cases, we may need to add a type hint like the following:

```
Domino = Union[Domino_1, Domino_2]
```

This definition provides a type name, `Domino`, with two concrete implementations. This provides information `mypy` can use to validate our software without the needless complexity of creating an abstract superclass. We can introduce new classes to this `Union` type without having to worry about inheritance. In order for this to work, the only requirement is for the classes to support the methods actually used by the application.

With this definition, we can use the following kind of factory method for building `Domino` instances:

```
class DominoBoneYard:

    domino_class: Type[Domino] = Domino_1

    def __init__(self, limit: int = 6) -> None:
        self._dominoes: List[Domino] = [
            self.domino_class(x, y)
            for x in range(limit + 1)
                for y in range(x + 1)
        ]
```

```
        random.shuffle(self._dominoes)
```

The `__init__()` method builds the `self._dominoes` object with a type hint of `List[Domino]`. This hint embraces all of the classes in the `Union[]` type hint for the `Domino` type name.

If we were to make a terrible mistake in using this class and try to use some code like `DominoBoneYard.domino_class = tuple` to create `tuple` objects, the `mypy` program would spot the type incompatibility and report an error with a message along the lines of `Incompatible types in assignment (expression has type "Type[Tuple[Any, ...]]", variable has type "Union[Type[Domino_1], Type[Domino_2]]")`. This message would inform us that the configuration choice of **tuple** is unlikely to work correctly.

Parallels between Python and libstdc++

The C++ Standard Template Library provides a number of design patterns realized as class templates. These templates must be filled in with specific types in order to satisfy the C++ compiler requirements. We can use this library as a suggestion for common design patterns. We'll look at a few elements of the modern GNU libstdc++ implementation as a representative sample of current thinking from other languages. The website at `https://en.cppreference.com/w/` provides a thorough reference.

The intent here is to use this library as a list of suggestions or hints about design patterns. This can provide a perspective on the features available in Python. There is no single, correct, *gold standard* for class libraries. Any comparison among languages is fraught with difficulties because it can appear as if one language is deficient because of a missing feature. In all cases when comparing languages, any *missing* feature can be trivially built from existing components.

An overview of various classes and templates in C++ library is available in chapters 4 to 15 of *The GNU C++ Library Manual*. Here is a comparison of some common patterns from another language and the mapping to similar features available in Python; they are as follows:

- **Support** describes some foundational features, including the atomic data types. The types defined here parallel `int`, `float`, and `str` in Python.
- **Diagnostics** describes the C++ implementation of exceptions and error handling.
- **Utilities** describes Functors and Pairs, which correspond to Python functions and two-tuples. Python extends the Pair concept to the more general `tuple` and adds `NamedTuple`.
- **Strings** describes more string features. Some of this is part of the `str` type, the `string` module, and the `shlex` module.
- **Localization** describes some additional features of the C++ libraries for localization. This is part of the Python `locale` module.
- **Containers** describes a number of C++ container class templates. We'll

provide details in a following section.

- **Iterators** describes the C++ features similar to Python iterator. The C++ implementations include a variety of iterator categories. A random access iterator, for example, is similar to integer index values for a list. A forward iterator parallels the Python implementation of iterators. An input or output iterator in C++ is similar to iterable file-like objects in Python. In Python, a class that offers the `__iter__()` method is iterable. An object with the `__next__()` method is an iterator. There are bidirectional iterators available in C++ – it's not clear how important these are, but there is a place in the class hierarchy for them.
- **Algorithms** describes classes, and expands on features in the Iterators chapter. It provides some tools for parallelizing algorithms. Most importantly, it contains a number of common operations. Some of these are parts of Python `collections`, others are part of `itertools`, and others are Python built-in functions.
- **Numeric's** describes some more advanced numeric types. The Python `complex` type and the `numbers` module provide Python versions of these features. The `array` module and packages, such as `numpy` and `pandas`, amplify these core features.
- **Input and Output** describes the C++ I/O classes. The Python `io` module defines equivalents for these C++ features.
- **Atomics** provides a way to define thread-safe objects where writes to memory are guaranteed to be completed before another thread can read them. In Python, the `threading` library can be used to build objects with this behavior.
- **Concurrency** describes additional ways of handling threads in an application. This is also part of the Python `threading` library.

The containers library in C++ includes the following kinds of categories and container class definitions:

- **Sequences** include the following:
    - **Array**. This is for fixed-size arrays. We can use a built-in `list`, or the `array` module, or even the `numpy` package for this.
    - **Vector**. This corresponds to the Python `list` class.
    - **Deque**. While the `list` class has these features, the `Collections.deque` class provides better performance.
    - **List**. This also corresponds to the Python `list` class.
    - **Forward List**. This is an extension to a list that permits removing

elements while iterating through them. Because this isn't directly available in Python, we'll often use `list(filter(rule, data))` to create a new list that is a subset of the old list.

- **Associations** are essentially the same as the unordered associations listed in the following bullet point. In the C++ implementation, a tree data structure is used to keep keys in order. In Python, an equivalent set of features can be built by using `sorted()` on the keys.
- **Unordered Associations** include the following:
  - **Set**. This corresponds roughly to the Python `set` class.
  - **Multiset**. This corresponds to the Python `collections.Counter` class.
  - **Map**. This corresponds to the Python `dict` class.
  - **Multimap**. This can be built using `defaultdict(list)`. Several add-on packages provide implementations for this. See http://werkzeug.pocoo.org/docs/0.14/datastructures/ for the `MultiDict` class, as one example.

When we consider the C++ libraries as a kind of benchmark, it appears Python provides similar features. This is a way to judge the completeness of Python design patterns. This alternative organization of these classes can be helpful to visualize the variety of implementation patterns present in Python.

Summary

In this chapter, we took a step back from the details of Python to look at the SOLID design principles. These considerations are fundamental to how a stateful object should be designed. The principles provide us a useful collection of ideas for structuring an object-oriented design. It seems most useful to consider the principles in the following order:

- **Interface Segregation**: Build the smallest interface to each class, refactoring to split a big class definition into smaller pieces.
- **Liskov Substitution**: Be sure that any subclass can replace the parent class; otherwise, consider a composition technique instead of inheritance.
- **Open/Closed**: A class should be open to extension but closed to direct modification. This requires careful consideration of what extensions are sensible for a given class.
- **Dependency Inversion**: A class shouldn't have a simple, direct dependency on another class. A class should be provided via a variable so a runtime configuration can change the class used.
- **Single Responsibility**: This summarizes the objective of defining classes with a single, narrow purpose, so changes are confined to one or a very few classes.

These principles will be implicit in the following chapters. The next chapter, `Chapter 16`, *The Logging and Warning Modules*, will look at using the `logging` and `warnings` modules to create audit information as well as to debug. We'll look at designing for testability and how we use `unittest` and `doctest` in `Chapter 17`, *Designing for Testability*. Later chapters will look at the design of applications, packages, and the general concepts of producing high-quality software.

The Logging and Warning Modules

We often need to make internal object state and state transitions more visible. There are the following three common cases for increased visibility:

- One case is for auditing an application, where we want to keep a history of state changes for an object.
- Another situation is to track the secure operations of an application and identify who is carrying out sensitive actions.
- A third common situation is to help debug problems that arise during use.

The Python logger is a flexible way to make internal object states and state transitions visible.

There are times when we have multiple logs with different kinds of information. We might distribute security, audit, and debugging into separate logs. In other cases, we might want a unified log. The `logging` module permits a variety of configurations.

Some users may want verbose output to confirm that the program works as they understand it. Allowing them to set a verbosity level produces a variety of log details focused on the needs of users.

The `warnings` module can also provide helpful information for developers as well as users, including the following:

- In the case of developers, we may use warnings to show them that an API has been deprecated
- In the case of users, we might want to show them the results are questionable but not erroneous

There might be questionable assumptions or possibly confusing default values that should be pointed out to users.

Software maintainers will need to selectively enable logging to perform useful debugging. We rarely want *blanket* debugging output: the resulting log might be

unreadably dense. We often need focused debugging to track down a specific problem in a specific class or module. The idea of multiple logs being sent a single handler can be used to enable detailed logging in some places and summary logging in others. The logging package in the Python 3.7.2 version of the standard library doesn't have complete type hints. Consequently, the examples in this chapter won't have type details. In this chapter, we will cover the following topics:

- Creating a basic log
- Configuration Gotcha
- Specialized logging for control, debug, audit, and security
- Using the warnings module
- Advanced logging—the last few messages and network destinations

Technical requirements

The code files for this chapter are available at `https://git.io/fj2U1`.

Creating a basic log

There are three steps to producing a log. The two *necessary* steps are the following:

1. Get a `logging.Logger` instance with the `logging.getLogger()` function; for example, `logger=logging.getLogger("demo")`.
2. Create messages with that `Logger`. There are a number of methods, with names such as `warn()`, `info()`, `debug()`, `error()`, and `fatal()`, that create messages with different levels of importance. For example, `logger.info("hello world")`.

These two steps are not sufficient to give us any output, however. There's a third, optional step that we take when we want to see logged messages. The reason for having a third step is because seeing a log isn't always required. Consider a debugging log that is generally left silent. The optional step is to configure the `logging` module's handlers, filters, and formatters. We can use the `logging.basicConfig()` function for this; for example, `logging.basicConfig(stream=sys.stderr, level=logging.INFO)`.

It's technically possible to skip the first step. We can use the default logger, which is part of the `logging` module's top-level functions. We showed you this in `Chapter 9`, *Decorators and Mixins – Cross-Cutting Aspects*, because the focus was on decoration, not logging. It is advisable not to use the default root logger and suggest that it's universally more configurable to use named loggers that are children of the root logger.

Instances of the `Logger` class are identified by a name attribute. The names are dot-separated strings that form a hierarchy. There's a root logger with the name `""`, the empty string. All other `Logger` instances are children of this root `Logger` instance. A complex application named `foo` might have an internal package named `services` with a module named `persistence` and a class named `SQLStore`. This could lead to loggers named `""`, `"foo"`, `"foo.services"`, `"foo.services.persistence"`, and `"foo.services.persistence.SQLStore"`.

We can often use the root `Logger` to configure the entire tree of `Logger` instances. When we choose names to form a proper hierarchy, we can enable or disable

whole subtrees of related instances by configuring an appropriate parent `Logger` object. In our preceding example, we might enable debugging for `"foo.services.persistence"` to see messages from all of the classes with a common prefix for their logger name.

In addition to a name, each `Logger` object can be configured with a list of `Handler` instances that determines where the messages are written and a list of `Filter` objects to determine which kinds of messages are passed or rejected. These `Logger` instances have the essential API for logging; we use a `Logger` object to create `LogRecord` instances. These records are then routed to `Filter` and `Handler` objects; the passed records are formatted and eventually wind up getting stored in a local file, or transmitted over a network.

The best practice is to have a distinct logger for each of our classes or modules. As `Logger` object names are dot-separated strings, the `Logger` instance names can parallel class or module names; our application's hierarchy of component definitions will have a parallel hierarchy of loggers. We might have a class that starts like the following code:

```
import logging
class Player:

    def __init__(self, bet: str, strategy: str, stake: int) -> None:
        self.logger = logging.getLogger(
            self.__class__.__qualname__)
        self.logger.debug(
            "init bet %r, strategy %r, stake %r",
            bet, strategy, stake
        )
```

The unique value of `self.__class__.__qualname__` will ensure that the `Logger` object used for this class will have a name that matches the qualified name of the class.

As a general approach to logging, this works well. The only downside of this approach is that each logger instance is created as part of the object, a tiny redundancy. It would be a somewhat better use of memory to create the logger as part of the class instead of each instance of the class.

In the next section, we'll look at a few ways to create a class-level logger shared by all instances of the class.

Creating a class-level logger

As we noted in Chapter 9, *Decorators and Mixins – Cross-Cutting Aspects*, creating a class-level logger can be done with a decorator. This will separate logger creation from the rest of the class. A common decorator idea that is very simple has a hidden problem. Here's the example decorator:

```
def logged(cls: Type) -> Type:
    cls.logger = logging.getLogger(cls.__qualname__)
    return cls
```

The `@logged` decorator creates the `logger` attribute as a feature of a class. This can then be shared by all of the instances. With this decorator, we can define a class with code like the following example:

```
@logged
class Player_1:

    def __init__(self, bet: str, strategy: str, stake: int) -> None:
        self.logger.debug("init bet %s, strategy %s, stake %d", bet,
        strategy, stake)
```

This will assure us that the `Player_1` class has the logger with the expected name of `logger`. We can then use `self.logger` in the various methods of this class.

The problem with this design is `mypy` is unable to detect the presence of the `logger` instance variable. This gap will lead `mypy` to report potential problems. There are several better approaches to creating loggers.

We can create a class-level debugger using code like the following example:

```
class Player_2:
    logger = logging.getLogger("Player_2")

    def __init__(self, bet: str, strategy: str, stake: int) -> None:
        self.logger.debug("init bet %s, strategy %s, stake %d", bet, strategy, stake)
```

This is simple and very clear. It suffers from a small **Don't Repeat Yourself** (**DRY**) problem. The class name is repeated within the class-level logger creation. This is a consequence of the way classes are created in Python, and there is no easy way to provide the class name to an object created before the class exists. It's the job of the metaclass to do any finalization of the class

definition; this can include providing the class name to internal objects.

We can use the following design to build a consistent logging attribute in a variety of related classes:

```
class LoggedClassMeta(type):

    def __new__(cls, name, bases, namespace, **kwds):
        result = type.__new__(cls, name, bases, dict(namespace))
        result.logger = logging.getLogger(result.__qualname__)
        return result

class LoggedClass(metaclass=LoggedClassMeta):
    logger: logging.Logger
```

This metaclass uses the `__new__()` method to create the resulting object and add a logger to the class. As an example, a class named `C` will then have a `C.logger` object. The `LoggedClass` can be used as a mixin class to provide the visible `logger` attribute name and also be sure it's properly initialized.

We'll use this class as shown in the following example:

```
class Player_3(LoggedClass):

    def __init__(self, bet: str, strategy: str, stake: int) -> None:
        self.logger.debug(
            "init bet %s, strategy %s, stake %d",
            bet, strategy, stake)
```

When we create an instance of `Player_3`, we're going to exercise the `logger` attribute. Because this attribute is set by the metaclass for `LoggedClass`, it is dependably set for every instance of the `Player_3` class.

The metaclass and superclass pair is superficially complex-looking. It creates a shared class-level logger for each instance. The name of the class is not repeated in the code. The only obligation on the client is to include `LoggedClass` as a mixin.

By default, we won't see any output from a definition like this. The initial configuration for the `logging` module doesn't include a handler or a level that produces any output. We'll also need to change the `logging` configuration to see any output.

The most important benefit of the way the `logging` module works is that we can include logging features in our classes and modules without worrying about the overall configuration. The default behavior will be silent and introduce very little

overhead. For this reason, we can always include logging features in every class that we define.

Configuring loggers

There are the following two configuration details that we need to provide in order to see the output in our logs:

- The logger we're using needs to be associated with at least one handler that produces conspicuous output.
- The handler needs a logging level that will pass our logging messages.

The `logging` package has a variety of configuration methods. We'll show you `logging.basicConfig()` here. We'll take a look at `logging.config.dictConfig()` separately.

The `logging.basicConfig()` method permits a few parameters to create a single `logging.handlers.StreamHandler` for logging the output. In many cases, this is all we need:

```
>>> import logging
>>> import sys
>>> logging.basicConfig(stream=sys.stderr, level=logging.DEBUG)
```

This will configure a `StreamHandler` instance that will write to `sys.stderr`. It will pass messages that have a level that is greater than or equal to the given level. By using `logging.DEBUG`, we're assured of seeing all the messages. The default level is `logging.WARN`.

After performing the basic configuration, we'll see our debugging messages when we create the class, as follows:

```
>>> pc3 = Player_3("Bet3", "Strategy3", 3)
DEBUG:Player_3:init bet Bet3, strategy Strategy3, stake 3
```

The default log format shows us the level (`DEBUG`), the name of the logger (`Player_3`), and the string that we produced. There are more attributes in `LogRecord` that can also be added to the output. Often, this default format is acceptable.

Starting up and shutting down the logging system

The `logging` module is defined in a way that avoids manually managing the global state information. The global state is handled within the `logging` module. We can write applications in separate parts and be well assured that those components will cooperate properly through the `logging` interface. We can, for example, include `logging` in some modules and omit it entirely from other modules without worrying about the compatibility or configuration.

Most importantly, we can include logging requests throughout an application and never configure any handlers. A top-level main script can omit `import logging` entirely. In this case, logging is a stand-by feature, ready for use when needed for debugging.

Because of the decentralized nature of logging, it's best to configure it only once, at the top level of an application. We should configure `logging` inside the `if __name__ == "__main__":` portion of an application. We'll look at this in more detail in `Chapter 18`, *Coping with the Command Line*.

Many of our logging handlers involve buffering. For the most part, data will be flushed from the buffers in the normal course of events. While we can ignore how logging shuts down, it's slightly more reliable to use `logging.shutdown()` to be sure that all of the buffers are flushed to the devices.

When handling top-level errors and exceptions, we have two explicit techniques to ensure that all buffers are written. One technique is to use a `finally` clause on a `try:` block, as follows:

```
import sys

if __name__ == "__main__":
    logging.config.dictConfig(yaml.load("log_config.yaml"))
    try:
        application = Main()
        status = application.run()
    except Exception as e:
        logging.exception(e)
        status = 1
```

```
    finally:
        logging.shutdown()
    sys.exit(status)
```

This example shows us how we configure `logging` as early as possible and shut down `logging` as late as possible. This ensures that as much of the application as possible is properly bracketed by properly configured loggers. This includes an exception logger; in some applications, the `main()` function handles all exceptions, making the `except` clause here redundant.

Another approach is to include an `atexit` handler to shut down `logging`, as follows:

```
import atexit
import sys
if __name__ == "__main__":
    logging.config.dictConfig(yaml.load("log_config.yaml"))
    atexit.register(logging.shutdown)
    try:
        application = Main()
        status = application.run()
    except Exception as e:
        logging.exception(e)
        status = 2
    sys.exit(status)
```

This version shows us how to use the `atexit` handler to invoke `logging.shutdown()`. When the application exits, the given function will be called. If the exceptions are properly handled inside the `main()` function, the `try:` block can be replaced with the much simpler `status = main(); sys.exit(status)`.

There's a third technique, which uses a context manager to control logging. We'll look at that alternative in Chapter 18, *Coping with the Command Line*.

Naming loggers

There are four common use cases for using `logging.getLogger()` to name our `Loggers`. We often pick names to parallel our application's architecture, as described in the following examples:

- **Module names**: We might have a module global `Logger` instance for modules that contain a large number of small functions or classes for which a large number of objects are created. When we extend `tuple`, for example, we don't want a reference to `Logger` in each instance. We'll often do this globally, and usually close to the front of the module, as follows:

```
import logging
logger = logging.getLogger(__name__)
```

- **Object instances**: This was shown previously, when we created `Logger` in the `__init__()` method. This `Logger` will be unique to the instance; using only a qualified class name might be misleading, because there will be multiple instances of the class. A better design is to include a unique instance identifier in the logger's name, as follows:

```
def __init__(self, player_name)
    self.name = player_name
    self.logger = logging.getLogger(
        f"{self.__class__.__qualname__}.{player_name}")
```

- **Class names**: This was shown previously, when we defined a simple decorator. We can use `__class__.__qualname__` as the `Logger` name and assign `Logger` to the class as a whole. It will be shared by all instances of the class.
- **Function names**: For small functions that are used frequently, we'll often use a module-level log, as shown previously. For larger functions that are rarely used, we might create a log within the function, as follows:

```
def main():
    log = logging.getLogger("main")
```

The idea here is to be sure that our `Logger` names match the names of components in our software architecture. This provides us with the most transparent logging, simplifying debugging.

In some cases, however, we might have a more complex collection of `Loggers`. We might have several distinct types of informational messages from a class. Two common examples are financial audit logs and security access logs. We might want several parallel hierarchies of `Loggers`; one with names that start with `audit.` and another with names that start with `security.` A class might have more specialized `Loggers`, with names such as `audit.module.Class` or `security.module.Class`, as shown in the following example:

```
self.audit_log = logging.getLogger(
    f"audit.{self.__class__.__qualname__}")
```

Having multiple logger objects available in a class allows us to finely control the kinds of output. We can configure each `Logger` to have different `handlers`. We'll use the more advanced configurations in the following section to direct the output to different destinations.

Extending logger levels

The `logging` module has five predefined levels of importance. Each level has a global variable (or two) with the level number. The level of importance represents a spectrum of optionality, from debugging messages (rarely important enough to show) to critical or fatal errors (always important), as shown in the following table:

Logging module variable	Value
`DEBUG`	10
`INFO`	20
`WARNING` or `WARN`	30
`ERROR`	40
`CRITICAL` or `FATAL`	50

We can add additional levels for even more nuanced control over what messages are passed or rejected. For example, some applications support multiple levels of verbosity. Similarly, some applications include multiple levels of debugging details. We might want to add an additional, verbose output level, set to 15, for example. This fits between information and debugging. It can follow the pattern of informative messages without devolving to the details of a debugging log.

For ordinary, silent processing, we might set the logging level to `logging.WARNING` so that only warnings and errors are shown. For the first level of verbosity, we can set the level of `logging.INFO` to see informational messages. For the second level of verbosity, we might want to add a level with a value of 15 and set the root logger to include this new level.

We can use the following to define our new level of verbose messages:

```
logging.addLevelName(15, "VERBOSE")
logging.VERBOSE = 15
```

This code needs to be written prior to the configuration of the loggers. It would be part of a top-level, main script. We can use our new levels via the `Logger.log( )` method, which takes the level number as an argument, as follows:

```
self.logger.log(logging.VERBOSE, "Some Message")
```

While there's little overhead to add levels such as these, they can be overused. The subtlety is that a level conflates multiple concepts—visibility and erroneous behavior—into a single numeric code. The levels should be confined to a simple visibility or error spectrum. Anything more complex must be done via the `Logger` names or the actual `Filter` objects.

Defining handlers for multiple destinations

We have several use cases for sending the log output to multiple destinations, which are shown in the following bullet list:

- We might want duplicate logs to improve the reliability of operations.
- We might be using sophisticated `Filter` objects to create distinct subsets of messages.
- We might have different levels for each destination. We can use the debugging level to separate debugging messages from informational messages.
- We might have different handlers based on the `Logger` names to represent different foci.

Of course, we can also combine these different choices to create quite complex scenarios. In order to create multiple destinations, we must create multiple `Handler` instances. Each `Handler` might contain a customized `Formatter`; it could contain an optional level, and an optional list of filters that can be applied.

Once we have multiple `Handler` instances, we can bind one or more `Logger` objects to the desired `Handler` instances. A `Handler` object can have a level filter. Using this, we can have multiple handler instances; each can have a different filter to show different groups of messages based on the level. Also, we can explicitly create `Filter` objects if we need even more sophisticated filtering than the built-in filters, which only check the severity level.

While we can configure this through the `logging` module API, it's often more clear to define most of the logging details in a separate configuration file. One elegant way to handle this is to use YAML notation for a configuration dictionary. We can then load the dictionary with a relatively straightforward use of `logging.config.dictConfig(yaml.load(somefile))`.

The YAML notation is somewhat more compact than the notation accepted by `configparser`. The documentation for `logging.config` in the *Python standard library*

uses YAML examples because of their clarity. We'll follow this pattern.

Here's an example of a configuration file with two handlers and two families of loggers:

```
version: 1
handlers:
  console:
    class: logging.StreamHandler
    stream: ext://sys.stderr
    formatter: basic
  audit_file:
    class: logging.FileHandler
    filename: data/ch16_audit.log
    encoding: utf-8
    formatter: basic
formatters:
  basic:
    style: "{"
    format: "{levelname:s}:{name:s}:{message:s}"
loggers:
  verbose:
    handlers: [console]
    level: INFO
  audit:
    handlers: [audit_file]
    level: INFO
```

We defined two handlers: `console` and `audit_file`. The `console` is `StreamHandler` that is sent to `sys.stderr`. Note that we have to use a URI-style syntax of `ext://sys.stderr` to name an *external* Python resource. In this context, external means external to the configuration file. This complex string is mapped to the `sys.stderr` object. The `audit_file` is `FileHandler` that will write to a given file. By default, files are opened with a mode of `a` to append.

We also defined the formatter, named `basic`, with a format to produce messages that match the ones created when logging is configured via `basicConfig()`. If we don't use this, the default format used by `dictConfig()` only has the message text.

Finally, we defined two top-level loggers, `verbose` and `audit`. The `verbose` instance will be used by all the loggers that have a top-level name of `verbose`. We can then use a `Logger` name such as `verbose.example.SomeClass` to create an instance that is a child of `verbose`. Each logger has a list of handlers; in this case, there's just one element in each list. Additionally, we've specified the logging level for each logger.

Here's how we can load this configuration file:

```
import logging.config
import yaml
config_dict = yaml.load(config)
logging.config.dictConfig(config_dict)
```

We parsed the YAML text into `dict` and then used the `dictConfig()` function to configure the logging with the given dictionary. Here are some examples of getting loggers and writing messages:

```
verbose = logging.getLogger("verbose.example.SomeClass")
audit = logging.getLogger("audit.example.SomeClass")
verbose.info("Verbose information")
audit.info("Audit record with before and after state")
```

We created two `Logger` objects; one under the `verbose` family tree and the other under the `audit` family tree. When we write to the `verbose` logger, we'll see the output on the console. When we write to the `audit` logger, however, we'll see nothing on the console; the record will go to the file that is named in the configuration.

When we look at the `logging.handlers` module, we see a large number of handlers that we can leverage. By default, the `logging` module uses old-style `%` style formatting specifications. These are not like the format specifications for the `str.format()` method. When we defined our formatter parameters, we used `{` style formatting, which is consistent with `str.format()`.

Managing propagation rules

The default behavior for `Loggers` is for a logging record to propagate from the named `Logger` up through all parent-level `Logger` instances to the root `Logger` instance. We may have lower-level `Loggers` that have special behaviors and a root `Logger` that defines the default behavior for all `Loggers`.

Because logging records propagate, a root-level logger will *also* handle any log records from the lower-level `Loggers` that we define. If child loggers allow propagation, this will lead to duplicated output: first, there will be output from the child, and then the output when the log record propagates to the parent. If we want to avoid duplication, we must turn the propagation off for lower-level loggers when there are handlers on several levels.

Our previous example does not configure a root-level `Logger`. If some part of our application creates a logger with a name that doesn't start with `audit.` or `verbose.`, then that additional logger won't be associated with `Handler`. Either we need more top-level names or we need to configure a catch-all, root-level logger.

If we add a root-level logger to capture all these other names, then we have to be careful about propagation rules. Here's a modification to the configuration file:

```
loggers:
  verbose:
    handlers: [console]
    level: INFO
    propagate: False # Added
  audit:
    handlers: [audit_file]
    level: INFO
    propagate: False # Added
root: # Added
  handlers: [console]
  level: INFO
```

We turned propagation off for the two lower-level loggers, `verbose` and `audit`. We added a new root-level logger. As this logger has no name, this was done as a separate top-level dictionary named `root:` in parallel with the `loggers:` entry.

If we didn't turn propagation off in the two lower-level loggers, each `verbose` or `audit` record would have been handled twice. In the case of an audit log, double

handling may actually be desirable. The audit data would go to the console as well as the audit file.

What's important about the `logging` module is that we don't have to make any application changes to refine and control the logging. We can do almost anything required through the configuration file. As YAML is relatively elegant notation, we can encode a lot of capability very simply.

Configuration Gotcha

The `basicConfig()` method of logging is careful about preserving any loggers created before the configuration is made. The `logging.config.dictConfig()` method, however, has the default behavior of disabling any loggers created prior to configuration.

When assembling a large and complex application, we may have module-level loggers that are created during the `import` process. The modules imported by the main script could potentially create loggers before `logging.config` is created. Also, any global objects or class definitions might have loggers created prior to the configuration.

We often have to add a line such as this to our configuration file:

```
disable_existing_loggers: False
```

This will ensure that all the loggers created prior to the configuration will still propagate to the root logger created by the configuration.

Specialized logging for control, debugging, audit, and security

There are many kinds of logging; we'll focus on these four varieties:

- **Errors and Control**: Basic errors and the control of an application leads to a main log that helps users confirm that the program really is doing what it's supposed to do. This would include enough error information with which users can correct their problems and rerun the application. If a user enables verbose logging, it will amplify this main error and control the log with additional user-friendly details.
- **Debugging**: This is used by developers and maintainers; it can include rather complex implementation details. We'll rarely want to enable *blanket* debugging, but will often enable debugging for specific modules or classes.
- **Audit**: This is a formal confirmation that tracks the transformations applied to data so that we can be sure that processing was done correctly.
- **Security**: This can be used to show us who has been authenticated; it can help confirm that the authorization rules are being followed. It can also be used to detect some kinds of attacks that involve repeated password failures.

We often have different formatting and handling requirements for each of these kinds of logs. Also, some of these are enabled and disabled dynamically. The main error and control log is often built from non-debug messages. We might have an application with a structure like the following code:

```
from collections import Counter
from Chapter_16.ch16_ex1 import LoggedClass

class Main(LoggedClass):

    def __init__(self) -> None:
        self.counts: Counter[str] = collections.Counter()

    def run(self) -> None:
        self.logger.info("Start")

        # Some processing in and around the counter increments
        self.counts["input"] += 2000
        self.counts["reject"] += 500
        self.counts["output"] += 1500
```

```
        self.logger.info("Counts %s", self.counts)
```

We used the `LoggedClass` class to create a logger with a name that matches the class qualified name (`Main`). We've written informational messages to this logger to show you that our application started normally and finished normally. In this case, we used `Counter` to accumulate some balance information that can be used to confirm that the right amount of data was processed.

In some cases, we'll have more formal balance information displayed at the end of the processing. We might do something like this to provide a slightly easier-to-read display:

```
    for k in self.counts:
        self.logger.info(
            f"{k:.<16s} {self.counts[k]:>6,d}")
```

This version will show us the keys and values on separate lines in the log. The errors and control log often uses the simplest format; it might show us just the message text with little or no additional context. A `formatter` like this might be used:

```
formatters:
  control:
    style: "{"
    format: "{levelname:s}:{message:s}"
```

This configures `formatter` to show us the level name (`INFO`, `WARNING`, `ERROR`, or `CRITICAL`) along with the message text. This eliminates a number of details, providing just the essential facts for the benefit of the users. We called the formatter `control`.

In the following code, we associate the formatter with the handler:

```
handlers:
  console:
    class: logging.StreamHandler
    stream: ext://sys.stderr
    formatter: control
```

This will use the `control` formatter with the `console` handler.

It's important to note that loggers will be created when the `Main` class, is created. This is long before the logging configuration is applied. In order to be sure the loggers that are defined as part of the classes are honored properly, the configuration must include the following:

```
disable_existing_loggers: False
```

This will guarantee that loggers created as part of the class definition will be preserved when `logging.config.dictConfig()` is used to set the configuration.

Creating a debugging log

A debugging log is usually enabled by a developer to monitor a program under development. It's often narrowly focused on specific features, modules, or classes. Consequently, we'll often enable and disable loggers by name. A configuration file might set the level of a few loggers to `DEBUG`, leaving others at `INFO`, or possibly even a `WARNING` level.

We'll often design debugging information into our classes. Indeed, we might use the debugging ability as a specific quality feature for a class design. This may mean introducing a rich set of logging requests. For example, we might have a complex calculation for which the class state is essential information as follows:

```
from Chapter_16.ch16_ex1 import LoggedClass

class BettingStrategy(LoggedClass):
    def bet(self) -> int:
        raise NotImplementedError("No bet method")

    def record_win(self) -> None:
        pass

    def record_loss(self) -> None:
        pass

class OneThreeTwoSix(BettingStrategy):
    def __init__(self) -> None:
        self.wins = 0

    def _state(self) -> Dict[str, int]:
        return dict(wins=self.wins)

    def bet(self) -> int:
        bet = {0: 1, 1: 3, 2: 2, 3: 6}[self.wins % 4]
        self.logger.debug(f"Bet {self._state()}; based on {bet}")
        return bet

    def record_win(self) -> None:
        self.wins += 1
        self.logger.debug(f"Win: {self._state()}")

    def record_loss(self) -> None:
        self.wins = 0
        self.logger.debug(f"Loss: {self._state()}")
```

In these class definitions, we defined a superclass, `BettingStrategy`, which provides some features of a betting strategy. Specifically, this class defines methods for getting a bet amount, recording a win, or recording a loss. The idea behind this

class is the common fallacy that the modification of bets can somehow mitigate losses in a game of chance.

The concrete implementation, `OneThreeTwoSix`, created a `_state()` method that exposes the relevant internal state. This method is only used to support debugging. We've avoided using `self.__dict__` because this often has too much information to be helpful. We can then audit the changes to the `self._state` information in several places in our method functions.

In many of the previous examples, we relied on the logger's use of `%r` and `%s` formatting. We might have used a line like `self.logger.info("template with %r and %r", some_item, another_variable)`. This kind of line provides a message with fields that pass through the filters before being handled by the formatter. A great deal of control can be exercised over lines like this.

In this example, we've used `self.logger.debug(f"Win: {self._state()}")`, which uses an f-string. The logging package's filter and formatter cannot be used to provide fine-grained control over this output. In the cases of audit and security logs, the logging `%` style formatting, controlled by the logger, is preferred. It allows a log filter to redact sensitive information in a consistent way. For informal log entries, f-strings can be handy. It's important, however, to be very careful of what information is placed in a log using f-string.

Debugging output is often selectively enabled by editing the configuration file to enable and disable debugging in certain places. We might make a change such as this to the logging configuration file:

```
loggers:
    betting.OneThreeTwoSix:
       handlers: [console]
       level: DEBUG
       propagate: False
```

We identified the logger for a particular class based on the qualified name for the class. This example assumes that there's a handler named `console` already defined. Also, we've turned off the propagation to prevent the debugging messages from being duplicated into the root logger.

Implicit in this design is the idea that debugging is not something we want to simply enable from the command line via a simplistic `-D` option or a `--DEBUG`

option. In order to perform effective debugging, we'll often want to enable selected loggers via a configuration file. We'll look at command-line issues in Chapter 18, *Coping with the Command Line*.

Creating audit and security logs

Audit and security logs are often duplicated between two handlers: the main control handler and a file handler that is used for audit and security reviews. This means that we'll do the following things:

- Define additional loggers for audit and security
- Define multiple handlers for these loggers
- Optionally, define additional formats for the audit handler

As shown previously, we'll often create separate hierarchies of the `audit` and `security` logs. Creating separate hierarchies of loggers is considerably simpler than trying to introduce audit or security via a new logging level. Adding new levels is challenging because the messages are essentially `INFO` messages; they don't belong on the `WARNING` side of `INFO` because they're not errors, nor do they belong on the `DEBUG` side of `INFO` because they're not optional.

Here's an extension to the metaclass shown earlier. This new metaclass will build a class that includes an ordinary control or debugging logger as well as a special auditing logger:

```
from Chapter_16.ch16_ex1 import LoggedClassMeta

class AuditedClassMeta(LoggedClassMeta):

    def __new__(cls, name, bases, namespace, **kwds):
        result = LoggedClassMeta.__new__(cls, name, bases, dict(namespace))
        for item, type_ref in result.__annotations__.items():
            if issubclass(type_ref, logging.Logger):
                prefix = "" if item == "logger" else f"{item}."
                logger = logging.getLogger(
                    f"{prefix}{result.__qualname__}")
                setattr(result, item, logger)
        return result


class AuditedClass(LoggedClass, metaclass=AuditedClassMeta):
    audit: logging.Logger
    pass
```

The `AuditedClassMeta` definition extends `LoggedClassMeta`. The base metaclass initialized the logged attribute with a specific logger instance based on the class name. This extension does something similar. It looks for all type annotations

that reference the `logging.Logger` type. All of these references are automatically initialized with a class-level logger with a name based on the attribute name and the qualified class name. This lets us build an audit logger or some other specialized logger with nothing more than a type annotation.

The `AuditedClass` definition extends the `LoggedClass` definition to provide a definition for a `logger` attribute of the class. This class adds the `audit` attribute to the class. Any subclass will be created with two loggers. One logger has a name simply based on the qualified name of the class. The other logger uses the qualified name, but with a prefix that puts it in the `audit` hierarchy. Here's how we can use this class:

```
class Table(AuditedClass):

    def bet(self, bet: str, amount: int) -> None:
        self.logger.info("Betting %d on %s", amount, bet)
        self.audit.info("Bet:%r, Amount:%r", bet, amount)
```

We created a class that will produce records on a logger with a name that has a prefix of `'audit.'`. The idea is to have two separate streams of log records from the application. In the main console log, we might want to see a simplified view, like the following example records:

```
INFO:Table:Betting 1 on Black
INFO:audit.Table:Bet:'Black', Amount:1
```

In the detailed audit file, however, we want more information, as shown in the following example records:

```
INFO:audit.Table:2019-03-19 07:34:58:Bet:'Black', Amount:1
INFO:audit.Table:2019-03-19 07:36:06:Bet:'Black', Amount:1
```

There are two different handlers for the `audit.Table` records. Each handler has a different format. We can configure logging to handle this additional hierarchy of loggers. We'll look at the two handlers that we need, as follows:

```
handlers:
  console:
    class: logging.StreamHandler
    stream: ext://sys.stderr
    formatter: basic
  audit_file:
    class: logging.FileHandler
    filename: data/ch16_audit.log
    encoding: utf-8
    formatter: detailed
```

The `console` handler has the user-oriented log entries, which use the `basic` format. The `audit_file` handler uses a more complex formatter named `detailed`. Here are the two `formatters` referenced by these `handlers`:

```
formatters:
  basic:
    style: "{"
    format: "{levelname:s}:{name:s}:{message:s}"
  detailed:
    style: "{"
    format: "{levelname:s}:{name:s}:{asctime:s}:{message:s}"
    datefmt: "%Y-%m-%d %H:%M:%S"
```

The `basic` format shows us just three attributes of the message. The `detailed` format rules are somewhat complex because the date formatting is done separately from the rest of the message formatting. The `datetime` module uses `%` style formatting. We used `{` style formatting for the overall message. Here are the two `Logger` definitions:

```
loggers:
  audit:
    handlers: [console,audit_file]
    level: INFO
    propagate: True
root:
  handlers: [console]
  level: INFO
```

We defined a logger for the `audit` hierarchy. All the children of `audit` will write their messages to both `console Handler` and `audit_file Handler`. The root logger will define all the other loggers to use the console only. We'll now see two forms of the audit messages.

The duplicate handlers provide us with the audit information in the context of the main console log, plus a focused audit trail in a separate log that can be saved for later analysis.

Using the warnings module

Object-oriented development often involves performing a significant refactoring of a class or module. It's difficult to get the API exactly right the very first time we write an application. Indeed, the design time required to get the API exactly right might get wasted. Python's flexibility permits us great latitude in making changes, as we learn more about the problem domain and the user's requirements.

One of the tools that we can use to support the design evolution is the `warnings` module. There are the following two clear use cases for `warnings` and one fuzzy use case:

- Warnings should be used to alert developers of API changes; usually, features that are deprecated or pending deprecation. Deprecation and pending deprecation warnings are silent by default. These messages are not silent when running the `unittest` module; this helps us ensure that we're making proper use of upgraded library packages.
- Warnings should alert users about a configuration problem. For example, there might be several alternative implementations of a module; when the preferred implementation is not available, we might want to provide a warning that an optimal implementation is not being used.
- We might push the edge of the envelope by using warnings to alert users that the results of the computation may have a problem. From outside the Python environment, one definition of a warning says, *...indicate that the service might have performed some, but not all, of the requested functions*. This idea of an *incomplete* result leading to a warning is open to dispute: it may be better to produce no result rather than a *potentially* incomplete result.

For the first two use cases, we'll often use Python's `warnings` module to show you that there are correctable problems. For the third blurry use case, we might use the `logger.warn()` method to alert the user about the potential issues. We shouldn't rely on the `warnings` module for this, because the default behavior is to show a warning just once.

The value of the warnings module is to provide messages that are optional and aimed at optimizations, compatibility, and a small set of runtime questions. The use of experimental features of a complex library or package, for example, might lead to a warning.

Showing API changes with a warning

When we change the API for one of our modules, packages, or classes, we can provide a handy marker via the `warnings` module. This will raise a warning in the method that is deprecated or is pending deprecation, as follows:

```
import warnings


class Player:
    """version 2.1"""

    def bet(self) -> None:
        warnings.warn(
            "bet is deprecated, use place_bet",
            DeprecationWarning, stacklevel=2)
        pass
```

When we do this, any part of the application that uses `Player.bet()` will receive `DeprecationWarning`. By default, this warning is silent. We can, however, adjust the `warnings` filter to see the message, as shown here:

```
>>> warnings.simplefilter("always", category=DeprecationWarning)
>>> p2 = Player()
>>> p2.bet() __main__:4: DeprecationWarning: bet is deprecated, use
    place_bet
```

This technique allows us to locate all of the places where our application must change because of an API change. If we have unit test cases with close to 100 percent code coverage, this simple technique is likely to reveal all the uses of deprecated methods.

Some **integrated development environments** (**IDEs**) can spot the use of warnings and highlight the deprecated code. PyCharm, for example, will draw a small line through any use of the deprecated `bet()` method.

Because this is so valuable for planning and managing software changes, we have the following three ways to make the warnings visible in our applications:

- The command-line `-Wd` option will set the action to `default` for all warnings. This will enable the normally silent deprecation warnings. When we run `python3.7 -Wd`, we'll see all the deprecation warnings.

- Using `unittest`, which always executes in the `warnings.simplefilter('default')` mode.
- Including `warnings.simplefilter('default')` in our application program. This will also apply the `default` action to all warnings; it's equivalent to the `-Wd` command-line option.

Showing configuration problems with a warning

We may have multiple implementations for a given class or module. We'll often use a configuration file parameter to decide which implementation is appropriate. See Chapter 14, *Configuration Files and Persistence*, for more information on this technique.

In some cases, however, an application may silently depend on whether or not other packages are part of the Python installation. One implementation may be optimal, and another implementation may be the fallback plan. This is used by a number of Python library modules to choose between optimized binary modules and pure Python modules.

A common technique is to try multiple `import` alternatives to locate a package that's installed. We can produce warnings that show us the possible configuration difficulties. Here's a way to manage this alternative implementation import:

```
import warnings

try:
    import simulation_model_1 as model
except ImportError as e:
    warnings.warn(repr(e))
if 'model' not in globals():
    try:
        import simulation_model_2 as model
    except ImportError as e:
        warnings.warn(repr(e))
if 'model' not in globals():
    raise ImportError("Missing simulation_model_1 and simulation_model_2")
```

We tried one import for a module. If this had failed, we'd have tried another import. We used an `if` statement to reduce the nesting of exceptions. If there are more than two alternatives, nested exceptions can lead to a very complex-looking exception. By using extra `if` statements, we can flatten a long sequence of alternatives so that the exceptions aren't nested.

We can better manage this warning message by changing the class of the message. In the preceding code, this will be `UserWarning`. These are shown by

default, providing users with some evidence that the configuration is not optimal.

If we change the class to `ImportWarning`, it will be silent by default. This provides a normally silent operation in cases where the choice of package doesn't matter to users. The typical developer's technique of running with the `-Wd` option will reveal the `ImportWarning` messages.

To change the class of the warning, we change the call to `warnings.warn()`, as follows:

```
warnings.warn(e, ImportWarning)
```

This changes the warning to a class that is silent by default. The message can still be visible to developers, who should be using the `-Wd` option.

Showing possible software problems with a warning

The idea of warnings aimed at end users is a bit nebulous; did the application work or did it fail? What does a warning really mean? Is there something the user should do differently?

Because of this potential ambiguity, warnings in the user interface aren't a great idea. To be truly usable, a program should either work correctly or should not work at all. When there's an error, the error message should include advice for the user's response to the problem. We shouldn't impose a burden on the user to judge the quality of the output and determine its fitness for purpose. We'll elaborate on this point.

> *A program should either work correctly or it should not work at all.*

One potential unambiguous use for end user warnings is to alert the user that the output is incomplete. An application may have a problem completing a network connection, for example; the essential results are correct, but one of the data sources didn't work properly.

There are situations where an application takes an action that is not what the user requested, and the output is valid and useful. In the case of a network problem, a default behavior can be used in spite of the network problem. Generally, replacing something faulty with something correct but not exactly what the user requested is a good candidate for a warning. This kind of warning is best done with `logging` at the WARN level, not with the `warnings` module. The `warnings` module produces one-time messages; we may want to provide more details to the user. Here's how we might use a simple `Logger.warn()` message to describe the problem in the log:

```
try:
    with urllib.request.urlopen("http://host/resource/", timeout=30) as resource:
        content = json.load(resource)
except socket.timeout as e:
    self.log.warn(
        "Missing information from  http://host/resource")
```

```
|     content= []
```

If a timeout occurs, a warning message is written to the log and the program keeps running. The content of the resource will be set to an empty list. The log message will be written every time. A `warnings` module warning is ordinarily shown only once from a given location in the program and is suppressed after that.

Advanced logging – the last few messages and network destinations

We'll look at two more advanced techniques that can help provide useful debugging information. The first of these is a *log tail;* this is a buffer of the last few log messages before some significant event. The idea is to have a small file that can be read to show why an application died. It's a bit like having the OS `tail` command automatically applied to the full log output.

The second technique uses a feature of the logging framework to send log messages through a network to a centralized log-handling service. This can be used to consolidate logs from a number of parallel web servers. We need to create both senders and receivers for the logs.

Building an automatic tail buffer

The log tail buffer is an extension to the `logging` framework. We're going to extend `MemoryHandler` to slightly alter its behavior. The built-in behavior for `MemoryHandler` includes three use cases for writing—it will write to another `handler` when the capacity is reached; it will write any buffered messages when `logging` shuts down; most importantly, it will write the entire buffer when a message of a given level is logged.

We'll change the first use case slightly. Instead of writing to the output file when the buffer is full, we'll remove the oldest messages, leaving the others in the buffer. The other two use cases for writing on exit and writing when a high-severity record is handled will be left alone. This will have the effect of dumping the last few messages before the shutdown, as well as dumping the last few messages before an error.

The default configuration for a `MemoryHandler` instance is to buffer messages until a message greater than or equal to the `ERROR` level is logged. This will lead to dumping the buffer when logging an error. It will tend to silence the *business as usual* messages that aren't immediate precursors of the error.

In order to understand this example, it's important to locate your Python installation and review the `logging.handlers` module in detail.

This extension to `MemoryHandler` will keep the last few messages, based on the defined capacity when the `TailHandler` class is created, as follows:

```
class TailHandler(logging.handlers.MemoryHandler):
    def shouldFlush(self, record: logging.LogRecord) -> bool:
        """
        Check for buffer full
        or a record at the flushLevel or higher.
        """
        if record.levelno >= self.flushLevel:
            return True
        while len(self.buffer) > self.capacity:
            self.acquire()
            try:
                del self.buffer[0]
            finally:
                self.release()
        return False
```

We extended `MemoryHandler` so that it will accumulate log messages up to the given capacity. When the capacity is reached, old messages will be removed as new messages are added. Note that we must lock the data structure to permit multithreaded logging.

If a message with an appropriate level is received, then the entire structure is emitted to the target handler. Usually, the target is `FileHandler`, which writes to a tail file for debugging and support purposes.

Additionally, when logging shuts down, the final few messages will also be written to the tail file. This should indicate a normal termination that doesn't require any debugging or support.

Generally, we'd send `DEBUG` level messages to this kind of handler so that we have a great deal of detail surrounding a crash situation. The configuration should specifically set the level to `DEBUG` rather than allowing the level to default.

Here's a configuration that uses this `TailHandler`:

```
version: 1
disable_existing_loggers: False
handlers:
  console:
    class: logging.StreamHandler
    stream: ext://sys.stderr
    formatter: basic
  tail:
    (): __main__.TailHandler
    target: cfg://handlers.console
    capacity: 5
formatters:
  basic:
    style: "{"
    format: "{levelname:s}:{name:s}:{message:s}"
loggers:
  test:
    handlers: [tail]
    level: DEBUG
    propagate: False
root:
  handlers: [console]
  level: INFO
```

The definition of `TailHandler` shows us several additional features of the `logging` configuration. It shows us class references as well as other elements of the configuration file.

We referred to a customized class definition in the configuration. A label of `()`

specifies that the value should be interpreted as a module and class name. In this case, it is an instance of our `__main__.TailHandler` class. A label of `class` instead of `()` uses a module and class that are part of the `logging` package.

We referred to another logger that's defined within the configuration. `cfg://handlers.console` refers to the `console` handler defined within the `handlers` section of this configuration file. For demonstration purposes, we used the `StreamHandler` tail target, which uses `sys.stderr`. As noted previously, a better design might be using `FileHandler`, which targets a debugging file.

We created the `test` hierarchy of loggers that used our `tail` handler. The messages written to these loggers will be buffered and only shown on the error or shutdown.

Here's a demonstration script:

```
logging.config.dictConfig(yaml.load(config8))
log = logging.getLogger("test.demo8")

log.info("Last 5 before error")
for i in range(20):
    log.debug(f"Message {i:d}")
log.error("Error causes dump of last 5")

log.info("Last 5 before shutdown")
for i in range(20, 40):
    log.debug(f"Message {i:d}")
log.info("Shutdown causes dump of last 5")

logging.shutdown()
```

We generated 20 messages prior to an error. Then, we generated 20 more messages before shutting down the logging and flushing the buffers. This will produce output like the following:

```
DEBUG:test.demo8:Message 15
DEBUG:test.demo8:Message 16
DEBUG:test.demo8:Message 17
DEBUG:test.demo8:Message 18
DEBUG:test.demo8:Message 19
ERROR:test.demo8:Error causes dump of last 5
DEBUG:test.demo8:Message 36
DEBUG:test.demo8:Message 37
DEBUG:test.demo8:Message 38
DEBUG:test.demo8:Message 39
INFO:test.demo8:Shutdown causes dump of last 5
```

The intermediate messages were silently dropped by the `TailHandler` object. As the capacity was set to five, the last five messages prior to an error (or shutdown) are

displayed. The last five messages include four debug messages plus a final informational message.

Sending logging messages to a remote process

One high-performance design pattern is to have a cluster of processes that are being used to solve a single problem. We might have an application that is spread across multiple application servers or multiple database clients. For this kind of architecture, we often want a centralized log among all of the various processes.

One technique for creating a unified log is to include accurate timestamps and then sort records from separate log files into a single, unified log. This sorting and merging is extra processing that can be avoided. Another, more responsive technique is to send log messages from a number of concurrent producer processes to a single consumer process.

Our shared logging solution makes use of the shared queues from the `multiprocessing` module. For additional information on multiprocessing, see `Chapter 13`, *Transmitting and Sharing Objects*.

The following is the three-step process to build a multiprocessing application:

- Firstly, we'll create a queue object shared by producers and consumers.
- Secondly, we'll create the consumer process, which gets the logging records from the queue. The logging consumer can apply filters to the messages and write them to a unified file.
- Thirdly, we'll create the pool of producer processes that do the real work of our application and produce logging records in the queue they share with the consumer.

As an additional feature, the `ERROR` and `FATAL` messages can provide immediate notification via an SMS or email to concerned users. The consumer can also handle the (relatively) slow processing associated with rotating log files.

Here's the definition of a consumer process:

```
import collections
```

```
import logging
import multiprocessing

class Log_Consumer_1(multiprocessing.Process):

    def __init__(self, queue):
        self.source = queue
        super().__init__()
        logging.config.dictConfig(yaml.load(consumer_config))
        self.combined = logging.getLogger(f"combined.{self.__class__.__qualname__}")
        self.log = logging.getLogger(self.__class__.__qualname__)
        self.counts = collections.Counter()

    def run(self):
        self.log.info("Consumer Started")
        while True:
            log_record = self.source.get()
            if log_record == None:
                break
            self.combined.handle(log_record)
            self.counts[log_record.getMessage()] += 1
        self.log.info("Consumer Finished")
        self.log.info(self.counts)
```

This process is a subclass of `multiprocessing.Process`. The multiprocessing.Process class provides a `start()` method, which will fork a subprocess and executes the `run()` method provided here.

The self.counts object tracks individual messages from the producers. The idea here is to create a summary showing the types of messages received. This is not a common practice, but it helps here to reveal how the demonstration works.

While the process is running, this object will use the `Queue.get()` method to get the log records from the queue. The messages will be routed to a logger instance. In this case, we're going to create a special logger named with a parent name of `combined.`; this will be given each record from a source process.

A sentinel object, `None`, will be used to signal the end of processing. When this is received, the `while` statement will finish and the final log messages will be written. The `self.counts` object will demonstrate how many messages were seen. This lets us tune the queue size to be sure messages aren't lost due to queue overruns.

Here's a `logging` configuration file for this process:

```
version: 1
disable_existing_loggers: False
handlers:
  console:
    class: logging.StreamHandler
```

```
    stream: ext://sys.stderr
    formatter: basic
formatters:
  basic:
    style: "{"
    format: "{levelname:s}:{name:s}:{message:s}"
loggers:
  combined:
    handlers: [console]
    formatter: detail
    level: INFO
    propagate: False
root:
  handlers: [console]
  level: INFO
```

We defined a simple console `Logger` with a basic format. We also defined the top-level of a hierarchy of loggers with names that begin with `combined.`. These loggers will be used to display the combined output of the various producers.

Here's a logging producer:

```
import multiprocessing
import time
import logging
import logging.handlers

class Log_Producer(multiprocessing.Process):
    handler_class = logging.handlers.QueueHandler

    def __init__(self, proc_id, queue):
        self.proc_id = proc_id
        self.destination = queue
        super().__init__()
        self.log = logging.getLogger(
            f"{self.__class__.__qualname__}.{self.proc_id}")
        self.log.handlers = [self.handler_class(self.destination)]
        self.log.setLevel(logging.INFO)

    def run(self):
        self.log.info(f"Started")
        for i in range(100):
            self.log.info(f"Message {i:d}")
            time.sleep(0.001)
        self.log.info(f"Finished")
```

The producer doesn't do much in the way of configuration. It gets a logger to use the qualified class name and an instance identifier (`self.proc_id`). It sets the list of handlers to be just `QueueHandler` wrapped around the destination—a `Queue` instance. The level of this logger is set to `INFO`.

We made `handler_class` an attribute of the class definition because we plan to change it. For the first example, it will be `logging.handlers.QueueHandler`. It allows the example producer to be reused with other kinds of handlers.

The process to actually do this work uses the logger to create log messages. These messages will be enqueued for processing by the centralized consumer. In this case, the process simply floods the queue with 102 messages as quickly as possible.

Here's how we can start the consumer and producers. We'll show this in small groups of steps. First, we create the queue as follows:

```
import multiprocessing
queue= multiprocessing.Queue(10)
```

This queue is way too small to handle 10 producers blasting 102 messages in a fraction of a second. The idea of a small queue is to see what happens when messages are lost. Here's how we start the consumer process:

```
consumer = Log_Consumer_1(queue)
consumer.start()
```

Here's how we start an array of producer processes:

```
producers = []
for i in range(10):
    proc= Log_Producer(i, queue)
    proc.start()
    producers.append(proc)
```

As expected, 10 concurrent producers will overflow the queue. Each producer will receive a number of queues full of exceptions to show us that the messages were lost.

Here's how we cleanly finish the processing:

```
for p in producers:
    p.join()
queue.put(None)
consumer.join()
```

First, we wait for each producer process to finish and then rejoin the parent process. Then, we put a sentinel object into the queue so that the consumer will terminate cleanly. Finally, we wait for the consumer process to finish and join the parent process.

Preventing queue overrun

The default behavior of the logging module puts messages into the queue with the `Queue.put_nowait()` method. The advantage of this is that it allows the producers to run without the delays associated with logging. The disadvantage of this is that messages will get lost if the queue is too small to handle a very large burst of logging messages.

We have the following two choices to gracefully handle a burst of messages:

- We can switch from `Queue` to `SimpleQueue`. `SimpleQueue` has an indefinite size. As it has a slightly different API, we'll need to extend `QueueHandler` to use `Queue.put()` instead of `Queue.put_nowait()`.
- We can slow down the producer in the rare case that the queue is full. This is a small change to `QueueHandler` to use `Queue.put()` instead of `Queue.put_nowait()`.

Interestingly, the same API change works for both `Queue` and `SimpleQueue`. Here's the change:

```
class WaitQueueHandler(logging.handlers.QueueHandler):

    def enqueue(self, record):
        self.queue.put(record)
```

We replaced the body of the `enqueue()` method to use a different method of `Queue`. Now, we can use `SimpleQueue` or `Queue`. If we use `Queue`, it will wait until the queue is full, preventing the loss of logging messages. If we use `SimpleQueue`, the queue will silently expand to hold all the messages.

Here's the revised producer class:

```
class Log_Producer_2(Log_Producer):
    handler_class = WaitQueueHandler
```

This class uses our new `WaitQueueHandler`. Otherwise, the producer is identical to the previous version.

The rest of the script to create `Queue` and start the consumer is identical. The producers are instances of `Log_Producer_2`, but otherwise, the script to start and join

remains identical to the first example.

This variation runs more slowly, but never loses a message. We can improve the performance by creating a larger queue capacity. If we create a queue with a capacity of 1,020 messages, the performance is maximized because that's the largest possible size of a burst. Finding an optimal queue capacity requires some experimentation. While the exact size depends on the operating system, a size of 30, for example, doesn't lose too many messages. The relative performance of the producer and consumer is important. To see the effects, change the sleep time in the producer to larger or smaller numbers. Also, it can be helpful to experiment by changing the number of producers from 10 to 100.

Summary

We saw how to use the logging module with more advanced object-oriented design techniques. We created logs associated with modules, classes, instances, and functions. We used decorators to create logging as a consistent cross-cutting aspect across multiple class definitions.

We saw how to use the `warnings` module to show you that there's a problem with the configuration or the deprecated methods. We can use warnings for other purposes, but we need to be cautious about the overuse of warnings and creating murky situations where it's not clear whether the application worked correctly or not.

Design considerations and trade-offs

The `logging` module supports auditability and debugging ability, as well as some security requirements. We can use logging as a simple way to keep records of the processing steps. By selectively enabling and disabling logging, we can support developers who are trying to learn what the code is really doing when processing real-world data.

The `warnings` module supports the debugging ability as well as maintainability features. We can use warnings to alert developers about API problems, configuration problems, and other potential sources of bugs.

When working with the `logging` module, we'll often be creating large numbers of distinct loggers that feed a few `handlers`. We can use the hierarchical nature of `Logger` names to introduce new or specialized collections of logging messages. There's no reason why a class can't have two loggers: one for audit and one for more general-purpose debugging.

We can introduce new logging level numbers, but this should be done reluctantly. Levels tend to conflate developer focus (debug, info, and warning) with user focus (info, error, and fatal). There's a kind of spectrum of *optionality* from debug messages that is not required for fatal error messages, which should never be silenced. We might add a level for verbose information or possibly detailed debugging, but that's about all that should be done with levels.

The `logging` module allows us to provide a number of configuration files for different purposes. As developers, we may use a configuration file that sets the logging levels to `DEBUG` and enables specific loggers for modules under development. For final deployment, we can provide a configuration file that sets the logging levels to `INFO` and provides different handlers to support more formal audit or security review needs.

We'll include some thoughts from *The Zen of Python* (`https://www.python.org/dev/peps/pep-0020/`):

"Errors should never pass silently.

Unless explicitly silenced."

The `warnings` and `logging` modules directly support this idea.

These modules are oriented more toward overall quality than toward the specific solution of a problem. They allow us to provide consistency via fairly simple programming. As our object-oriented designs become larger and more complex, we can focus more on the problem being solved without wasting time on the infrastructure considerations. Further, these modules allow us to tailor the output to provide information needed by the developer or user.

Looking ahead

In the following chapters, we'll take a look at designing for testability and how we use `unittest`, `doctest`, and the `pytest` package for testing. Automated testing is essential; no program should be considered complete until there are automated unit tests that provide ample evidence to show us that the code works. We'll also look at object-oriented design techniques that make software easier to test.

Designing for Testability

High-quality programs have automated tests. We need to use everything at our disposal to be sure that our software works. The golden rule is this: *to be deliverable, a feature must have an automated unit test*. Without an automated unit test, a feature cannot be trusted to work and should not be used. According to Kent Beck, in *Extreme Programming Explained*:

"Any program feature without an automated test simply doesn't exist."

There are the following two essential points regarding the automated testing of program features:

- **Automated**: This means that there's no human judgment involved. The testing involves a script that compares actual responses to expected responses.
- **Features**: Elements of the software are tested in isolation to be sure that they work separately. In some contexts, features are quite broad concepts involving functionality observed by the user. When unit testing, features are often much smaller, and refer to behaviors of single software components. Each *unit* is enough software to implement a given feature. Ideally, it's a Python class. However, it can also be a larger unit, such as a module or package.

Python has two built-in testing frameworks, making it easy to write automated unit tests. We'll look at using both `doctest` and `unittest` for automated testing. `doctest` offers a very simple approach to test writing. The `unittest` package is much more sophisticated.

We'll also look at the popular `pytest` framework, which is simpler to use than `unittest`. In some cases, we'll write tests using the `unittest.TestCase` class, but execute the tests using `pytest`'s sophisticated test discovery. In other cases, we'll do almost everything with `pytest`.

We'll look at some of the design considerations required to make testing practical. It's often necessary to decompose complex classes for testability. As

we noted in `Chapter 15`, *Design Principles and Patterns*, we want expose dependencies to have a flexible design; the dependency inversion principle will also help to create testable classes.

For more ideas, read about *Ottinger and Langr's* **FIRST** unit test properties —**Fast**, **Isolated**, **Repeatable**, **Self-Validating**, and **Timely**. For the most part, **Repeatable** and **Self-Validating** require an automated test framework. Timely means that the test is written before the code under test. For more information, refer to `http://pragprog.com/magazines/2012-01/unit-tests-are-first`.

In this chapter, we will cover the following topics:

- Defining and isolating units for testing
- Using `doctest` to define test cases
- Using setup and teardown
- The `TestCase` class hierarchy
- Using externally defined expected results
- Using `pytest` and `fixtures`
- Automated integration testing or automated performance testing

Technical requirements

The code files for this chapter are available at `https://git.io/fj2UM`.

Defining and isolating units for testing

When we consider testing to be essential, testability is an important design consideration for object-oriented programming. Our designs must also support testing and debugging, because a class that merely *appears* to work is of no value. A class that has evidence that it works is much more valuable.

Different types of tests form a kind of hierarchy. At the foundation of that hierarchy is unit testing. Here, we test each class or function in isolation to be sure that it meets the contractual obligations of the API. Each class or function is an isolated unit under test. Above this layer comes integration testing. Once we know that each class and function works individually, we can test groups and clusters of classes. We can test whole modules and whole packages too. Once the integration tests work, the next layer is the automated testing of the complete application. This is not an exhaustive list of the types of tests. We can do performance testing and security vulnerability testing too.

In this chapter, we'll focus on automated unit testing because it is central to all applications. The hierarchy of testing reveals important complexity. Test cases for an individual class or group of classes can be very narrowly defined. As we introduce more units into integration testing, the domain of inputs grows. When we attempt to test a whole application, the entire spectrum of human behavior becomes potential input. As we look more broadly, we see that it could include shutting devices off, pulling out plugs, or even pushing devices off tables to see whether they still work after being dropped three feet onto a hardwood floor. The hugeness of the domain of human behavior makes it difficult to *fully* automate all possible kinds of application testing. We'll focus on the things that are easiest to test automatically. Once the unit tests work, the larger aggregate systems are more likely to work.

Minimizing dependencies

When we design a class, we must also consider the network of dependencies around that class: classes on which it depends and classes that depend on it. In order to simplify testing a class definition, we need to isolate it from the surrounding classes. For more ideas on this, refer to `Chapter 15`, *Design Principles and Patterns*.

An example of this is the `Deck` class, which depends on the `Card` class. We can easily test `Card` in isolation, but when we want to test the `Deck` class, we need to tease it away from the definition of `Card`.

Here's one (of many) previous definition of `Card` that we've looked at:

```
import enum

class Suit(enum.Enum):
    CLUB = "♣"
    DIAMOND = "♦"
    HEART = "♥"
    SPADE = "♠"
class Card:
    def __init__(
        self, rank: int, suit: Suit, hard: int = None, soft: int = None
    ) -> None:
        self.rank = rank
        self.suit = suit
        self.hard = hard or int(rank)
        self.soft = soft or int(rank)
    def __str__(self) -> str:
        return f"{self.rank!s}{self.suit.value!s}"
class AceCard(Card):
    def __init__(self, rank: int, suit: Suit) -> None:
        super().__init__(rank, suit, 1, 11)
class FaceCard(Card):
    def __init__(self, rank: int, suit: Suit) -> None:
        super().__init__(rank, suit, 10, 10)
```

We can see that each of these classes has a straightforward inheritance dependency. Each class can be tested in isolation because there are only two methods and four attributes.

We can (mis-)design the `Deck` class. The following is a bad example, and has some problematic dependencies:

```
import random
```

```
class Deck1(list):

    def __init__(self, size: int=1) -> None:
        super().__init__()
        self.rng = random.Random()
        for d in range(size):
            for s in iter(Suit):
                cards: List[Card] = (
                    [cast(Card, AceCard(1, s))]
                    + [Card(r, s) for r in range(2, 12)]
                    + [FaceCard(r, s) for r in range(12, 14)]
                )
                super().extend(cards)
        self.rng.shuffle(self)
```

This design has two deficiencies. First, it's intimately bound to the three classes in the `Card` class hierarchy. We can't isolate `Deck` from `Card` for a standalone unit test. Second, it is dependent on the random number generator, making it difficult to create a repeatable test. When we look back at `Chapter 15`, *Design Principles and Patterns*, we can see these problems as a failure to follow the dependency inversion principle.

On the one hand, `Card` is a pretty simple class. We could test this version of `Deck` with `Card` left in place. On the other hand, we might want to reuse `Deck` with poker cards or pinochle cards, which have different behaviors from blackjack cards.

The ideal situation is to make `Deck` independent of any particular `Card` implementation. If we do this well, then we can not only test `Deck` independently of any `Card` implementation, but we can also use any combination of `Card` and `Deck` definitions.

Here's our preferred method to separate one of the dependencies. We can put these dependencies into a factory function, as shown in the following example:

```
class LogicError(Exception):
    pass

def card(rank: int, suit: Suit) -> Card:
    if rank == 1:
        return AceCard(rank, suit)
    elif 2 <= rank < 11:
        return Card(rank, suit)
    elif 11 <= rank < 14:
        return FaceCard(rank, suit)
    else:
        raise LogicError(f"Rank {rank} invalid")
```

The `card()` function will build proper subclasses of `Card` based on the requested

rank. This allows the `Deck` class to use this function instead of directly building instances of the `Card` class. We separated the two class definitions by inserting an intermediate function.

There are other techniques to separate the `Card` class from the `Deck` class. We can refactor the factory function to be a method of `Deck`. We can also make the class names a separate binding via class-level attributes or even initialization method parameters. Here's an example that avoids a factory function by using more complex bindings in the initialization method:

```
class Deck2(list):

    def __init__(
        self,
        size: int=1,
        random: random.Random=random.Random(),
        ace_class: Type[Card]=AceCard,
        card_class: Type[Card]=Card,
        face_class: Type[Card]=FaceCard,
    ) -> None:
        super().__init__()
        self.rng = random
        for d in range(size):
            for s in iter(Suit):
                cards = (
                    [ace_class(1, s)]
                    + [card_class(r, s) for r in range(2, 12)]
                    + [face_class(r, s) for r in range(12, 14)]
                )
                super().extend(cards)
        self.rng.shuffle(self)
```

While this initialization is wordy, the `Deck` class is no longer intimately bound to the `Card` class hierarchy or a specific random number generator. For testing purposes, we can provide a random number generator that has a known seed. We can also replace the various `Card` class definitions with other classes (such as `tuple`) that can simplify our testing.

Note the type hints for the default values provided to this class. The object provided for the `random` parameter should be an instance of the `random.Random` type, and the default value is an object of that exact type. Similarly, for the three card classes, each must be a `Type`, and a subclass of the `Card` class. The `Type[Card]` hint permits any class derived from `Card`. This permits `mypy` to confirm that overrides are likely to work. It can be difficult to check type hints in the test modules because, as of version 3.8.2, pytest doesn't have complete type hint stubs.

In the next section, we'll focus on another variation of the `Deck` class. This will

use the `card()` factory function. That factory function encapsulates the `Card` hierarchy bindings and the rules for separating card classes by rank into a single, testable location.

Creating simple unit tests

We'll create some simple unit tests of the `Card` class hierarchy and the `card()` factory function.

As the `Card` classes are so simple, there's no reason for overly sophisticated testing. It's always possible to err on the side of needless complication. An *unthinking* slog through a test-driven development process can make it seem like we need to write a fairly large number of not very interesting unit tests for a class that only has a few attributes and methods.

It's important to understand that test-driven development is *advice*, not a natural law like the conservation of mass. Nor is it a ritual that must be followed without thinking.

There are several schools of thought on naming test methods. We'll focus on a style of naming that includes describing a test condition and the expected results. Here are three variations on this theme:

- `StateUnderTest_should_ExpectedBehavior`
- `when_StateUnderTest_should_ExpectedBehavior`
- `UnitOfWork_StateUnderTest_ExpectedBehavior`

For more information, refer to:

http://osherove.com/blog/2005/4/3/naming-standards-for-unit-tests.html

The `StateUnderTest` portion of the name is often evident from the class that contains the test, and can be omitted from the method name. This means the individual test cases can emphasize the `should_ExpectedBehavior` portion of the name. In order to comply with the way the `unittest.TestCase` class works, each test behavior must begin with `test_`. This leads us to suggest `test_should_`*`ExpectedBehavior`* as a pattern for test case names when using individual test methods for a `unittest.TestCase` subclass. For `pytest` functions, we'll use a different pattern of naming.

It's possible to configure the `unittest` module to use different patterns for naming the test methods. We could change it to look for `when_` instead of `test_`. The improvement in names doesn't seem to be worth the effort required.

This, for example, is a test of the `Card` class:

```
class TestCard(unittest.TestCase):

    def setUp(self) -> None:
        self.three_clubs = Card(3, Suit.CLUB)

    def test_should_returnStr(self) -> None:
        self.assertEqual("3♣", str(self.three_clubs))

    def test_should_getAttrValues(self) -> None:
        self.assertEqual(3, self.three_clubs.rank)
        self.assertEqual(Suit.CLUB, self.three_clubs.suit)
        self.assertEqual(3, self.three_clubs.hard)
        self.assertEqual(3, self.three_clubs.soft)
```

We defined a test `setUp()` method that creates an object of the class that is being tested. We also defined two tests on this object. As there's no real interaction here, there's no *state under test* in the test names; they're simple universal behaviors that should always work.

This is a lot of test code for a very small class definition. This raises a question as to whether or not the volume of test code is in some way *excessive*. The answer is *no*; this is not an excessive amount of test code. There's no law that says that there should be more application code than test code. Indeed, it doesn't make sense to compare the volume of test code with the volume of application code. Most importantly, even a tiny class definition can still have bugs, and it may require complex tests to assure the bugs don't exist.

Simply testing the values of attributes doesn't seem to test the processing in this class. There are two perspectives on testing attribute values:

- The **black-box** perspective means that we disregard the implementation. In this case, we need to test all of the attributes. The attributes could, for example, be properties, and they must be tested.
- The **white-box** perspective means that we can examine the implementation details. When performing this style of testing, we can be a little more circumspect in which attributes we test. The `suit` attribute, for example, doesn't deserve much testing. The `hard` and `soft` attributes, however, do require testing.

For more information, refer to:

http://en.wikipedia.org/wiki/White-box_testing and http://en.wikipedia.org/wiki/Black-box_testing

Of course, we need to test the rest of the `Card` class hierarchy. We'll just show you the `AceCard` test case. The `FaceCard` test case should be clear after this example:

```
class TestAceCard(unittest.TestCase):

    def setUp(self) -> None:
        self.ace_spades = AceCard(1, Suit.SPADE)

    @unittest.expectedFailure
    def test_should_returnStr(self) -> None:
        self.assertEqual("A♠", str(self.ace_spades))

    def test_should_getAttrValues(self) -> None:
        self.assertEqual(1, self.ace_spades.rank)
        self.assertEqual(Suit.SPADE, self.ace_spades.suit)
        self.assertEqual(1, self.ace_spades.hard)
        self.assertEqual(11, self.ace_spades.soft)
```

This test case also sets up a particular `Card` instance so that we can test the string output. It checks the various attributes of this fixed card.

Note that the `test_should_returnStr()` test will fail. The definition of the `AceCard` class does not display the value as shown in this test definition. Either the test is incorrect or the class definition is incorrect. The unit test uncovered this fault in the class design.

A similar test is required for the `FaceCard` class. It will be similar to the test shown for the `AceCard` class. We won't present it here, but will leave it as an exercise for you.

When we have a number of tests, it can be helpful to combine them into a suite of tests. We'll turn to this next.

Creating a test suite

It is often helpful to formally define a test suite. The `unittest` package is capable of discovering tests by default. When aggregating tests from multiple test modules, it's sometimes helpful to create a test suite in every test module. If each module defines a `suite()` function, we can replace test discovery with importing the `suite()` functions from each module. Also, if we customize `TestRunner`, we must use a suite. We can execute our tests as follows:

```
def suite2() -> unittest.TestSuite:
    s = unittest.TestSuite()
    load_from = unittest.defaultTestLoader.loadTestsFromTestCase
    s.addTests(load_from(TestCard))
    s.addTests(load_from(TestAceCard))
    s.addTests(load_from(TestFaceCard))
    return s
```

We build a suite from our three `TestCase` class definitions and then provide that suite to a `unittest.TextTestRunner()` instance. We use the default `TestLoader` in `unittest`. This `TestLoader` examines a `TestCase` class to locate all the test methods. The value of `TestLoader.testMethodPrefix` is `test`, which is how test methods are identified within a class. Each method name is used by the loader to create a separate test object.

Using `TestLoader` to build test instances from appropriately named methods of `TestCase` is one of the two ways to use `TestCase`. In a later section, we'll look at creating instances of `TestCase` manually; we won't rely on `TestLoader` for these examples. We can run this suite with the following code:

```
if __name__ == "__main__":
    t = unittest.TextTestRunner()
    t.run(suite2())
```

We'll see output like the following code:

```
...F.F
======================================================================
FAIL: test_should_returnStr (__main__.TestAceCard)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "p3_c15.py", line 80, in test_should_returnStr
    self.assertEqual("A♠", str(self.ace_spades))
AssertionError: 'A♠' != '1♠'
- A♠
```

```
+ 1♠


======================================================================
FAIL: test_should_returnStr (__main__.TestFaceCard)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "p3_c15.py", line 91, in test_should_returnStr
    self.assertEqual("Q♥", str(self.queen_hearts))
AssertionError: 'Q♥' != '12♥'
- Q♥
+ 12♥


----------------------------------------------------------------------
Ran 6 tests in 0.001s

FAILED (failures=2)
```

The `TestLoader` class created two tests from each `TestCase` class. This gives us a total of six tests. The test names are the method names, which begin with `test`.

Clearly, we have a problem. Our tests provide an expected result that our class definitions don't meet. We've got more development work to do for the `Card` classes in order to pass this simple suite of unit tests. We'll leave the fixes as exercises for you.

Starting to design tests can seem daunting at first. There are a few guidelines that can be helpful. In the next section, we will talk about *edges*—often limits, and *corners*—often the interfaces between components, which can help us design better tests.

Including edge and corner cases

When we move to testing the `Deck` class as a whole, we'll need to have some things confirmed: that it produces all of the required `Cards` class, and that it actually shuffles properly. We don't really need to test that it deals properly because we're depending on the `list` and `list.pop()` method; as these are first-class parts of Python, they don't require additional testing.

We'd like to test the `Deck` class construction and shuffling independently of any specific `Card` class hierarchy. As noted previously, we can use a factory function to make the two `Deck` and `Card` definitions independent. Introducing a factory function introduces yet more testing. Not a bad thing, considering the bugs previously revealed in the `Card` class hierarchy.

Here's a test of the factory function:

```
class TestCardFactory(unittest.TestCase):

    def test_rank1_should_createAceCard(self) -> None:
        c = card(1, Suit.CLUB)
        self.assertIsInstance(c, AceCard)

    def test_rank2_should_createCard(self) -> None:
        c = card(2, Suit.DIAMOND)
        self.assertIsInstance(c, Card)

    def test_rank10_should_createCard(self) -> None:
        c = card(10, Suit.HEART)
        self.assertIsInstance(c, Card)

    def test_rank10_should_createFaceCard(self) -> None:
        c = card(11, Suit.SPADE)
        self.assertIsInstance(c, Card)

    def test_rank13_should_createFaceCard(self) -> None:
        c = card(13, Suit.CLUB)
        self.assertIsInstance(c, Card)

    def test_otherRank_should_exception(self) -> None:
        with self.assertRaises(LogicError):
            c = card(14, Suit.DIAMOND)
        with self.assertRaises(LogicError):
            c = card(0, Suit.DIAMOND)
```

We didn't test all 13 ranks, as 2 through 10 should all be identical. Instead, we followed this advice from Boris Beizer in the book *Software Testing Techniques*:

"Bugs lurk in corners and congregate at boundaries."

The test cases involve the edge values for each card range. Consequently, we have test cases for the values 1, 2, 10, 11, and 13, as well as the illegal values 0 and 14. We bracketed each range with the least value, the maximum value, one below the least value, and one above the maximum value.

We've modified the pattern for test naming. In this case, we have several different states being tested. We've modified the simpler names from these to followthe pattern `test_`*`StateUnderTest`*`_should_`*`ExpectedBehavior`*. There doesn't seem to be a compelling reason to break these tests into separate classes to decompose the state under test.

In the next section, we'll look at ways to handle dependent objects. This will allow us to test each unit in isolation.

Using mock objects to eliminate dependencies

In order to test `Deck`, we have the following two choices for handling the dependencies in the `Card` class hierarchy:

- **Mocking**: We can create a mock (or stand-in) class for the `Card` class and a mock `card()` factory function that produces instances of the mock class. The advantage of using mock objects is that we create real confidence that the unit under test is free from workarounds in one class, which make up for bugs in another class. A rare potential disadvantage is that we may have to debug the behavior of a super-complex mock class to be sure it's a valid stand-in for a real class. A complex mock object suggests the real object is too complex and needs to be refactored.
- **Integrating**: If we have a degree of trust that the `Card` class hierarchy works, and the `card()` factory function works, we can leverage these to test `Deck`. This strays from the high road of pure unit testing, in which all dependencies are omitted for test purposes. The disadvantage of this is that a broken foundational class will cause a large number of testing failures in all the classes that depend on it. Also, it's difficult to make detailed tests of API conformance with non-mock classes. Mock classes can track the call history, making it possible to track the details of calls to mock objects.

The `unittest` package includes the `unittest.mock` module which can be used to patch the existing classes for test purposes. It can also be used to provide complete mock class definitions. Later, when we look at `pytest`, we'll combine `unittest.mock` objects with the `pytest` test framework.

The examples in this section do not use extensive type hinting in the tests. For the most part, the tests should pass through `mypy` without difficulty. As we noted earlier, `pytest` version 3.8.2. doesn't have a complete set of type stubs, so the `--ignore-missing-imports` option must be used when running `mypy`. The *mock* objects, for the most part, provide type hints that permit `mypy` to confirm that they are being used properly.

When we design a class, we must consider the dependencies that must be mocked for unit testing. In the case of `Deck`, we have the following three dependencies to mock:

- **The `Card` class**: This class is so simple that we could create a mock for this class without basing it on an existing implementation. As the `Deck` class behavior doesn't depend on any specific feature of `Card`, our mock object can be simple.
- **The `card()` factory**: This function needs to be replaced with a mock that we can use to determine whether `Deck` makes proper calls to this function.
- **The `random.Random.shuffle()` method**: To determine whether the method was called with proper argument values, we can provide a mock that will track usage rather than actually doing any shuffling.

Here's a version of `Deck` that uses the `card()` factory function:

```
class DeckEmpty(Exception):
    pass

class Deck3(list):

    def __init__(
            self,
            size: int=1,
            random: random.Random=random.Random(),
            card_factory: Callable[[int, Suit], Card]=card
    ) -> None:
        super().__init__()
        self.rng = random
        for d in range(size):
            super().extend(
                [card_factory(r, s)
                 for r in range(1, 14)
                 for s in iter(Suit)])
        self.rng.shuffle(self)

    def deal(self) -> Card:
        try:
            return self.pop(0)
        except IndexError:
            raise DeckEmpty()
```

This definition has two dependencies that are specifically called out as arguments to the `__init__()` method. It requires a random number generator, `random`, and a card factory, `card_factory`. It has suitable default values so that it can be used in an application very simply. It can also be tested by providing mock objects instead of the default objects.

We've included a `deal()` method that makes a change to the object by using `pop()`

to remove an instance of `Card` from the collection. If the deck is empty, the `deal()` method will raise a `DeckEmpty` exception.

Here's a test case to show you that the deck is built properly:

```
import unittest
import unittest.mock

class TestDeckBuild(unittest.TestCase):

    def setUp(self) -> None:
        self.mock_card = unittest.mock.Mock(return_value=unittest.mock.sentinel.card)
        self.mock_rng = unittest.mock.Mock(wraps=random.Random())
        self.mock_rng.shuffle = unittest.mock.Mock()

    def test_Deck3_should_build(self) -> None:
        d = Deck3(size=1, random=self.mock_rng, card_factory=self.mock_card)
        self.assertEqual(52 * [unittest.mock.sentinel.card], d)
        self.mock_rng.shuffle.assert_called_with(d)
        self.assertEqual(52, len(self.mock_card.mock_calls))
        expected = [
            unittest.mock.call(r, s)
            for r in range(1, 14)
            for s in (Suit.CLUB, Suit.DIAMOND, Suit.HEART, Suit.SPADE)
        ]
        self.assertEqual(expected, self.mock_card.mock_calls)
```

We created two mocks in the `setUp()` method of this test case. The mock card factory function, `mock_card`, is a `Mock` function. The defined return value is a single `mock.sentinel,card` object instead of a distinct `Card` instances. When we refer to an attribute of the `mock.sentinel` object, with an expression like `mock.sentinel.card`, this expression creates a new object if necessary, or retrieves an existing object. It implements a *Singleton* design pattern; there's only one `sentinel` object with a given name. This will be a unique object that allows us to confirm that the right number of instances were created. Because the `sentinel` is distinct from all other Python objects, we can distinguish functions without proper `return` statements that return `None`.

We created a mock object, `mock_rng`, to wrap an instance of the `random.Random()` generator. This `Mock` object will behave as a proper random object, with one difference. We replaced the `shuffle()` method with a `Mock` behaving as a function that returns `None`. This provides us with an appropriate return value for the method and allows us to determine that the `shuffle()` method was called with the proper argument values.

Our test creates a `Deck3` instance with our two mock objects. We can then make the following assertions about the `Deck3` instance, `d`:

- 52 objects were created. These are expected to be 52 copies of `mock.sentinel`, showing us that only the factory function was used to create objects; all of the objects are sentinels, created by the mock.
- The `shuffle()` method was called with the `Deck` instance as the argument. This shows us how a mock object tracks its calls. We can use `assert_called_with()` to confirm that the argument values were as required when `shuffle()` was called.
- The factory function was called 52 times. The `mock_calls` attribute of a mock object is the entire history of the object's use. This assertion is, technically, redundant, since the next test will imply this condition.
- The factory function was called with a specific list of expected rank and suit values.

The mock objects will record the sequence of methods that were invoked. In the next section, we'll look at ways to examine a mock object to ensure that other units are using the mock object correctly.

Using mocks to observe behaviors

The preceding mock objects were used to test how a `Deck` class was built. Having 52 identical sentinels makes it difficult to confirm that a `Deck` deals properly. We'll define a different mock to test the deal feature.

Here's a second test case to ensure that the `Deck` class deals properly:

```
class TestDeckDeal(unittest.TestCase):

    def setUp(self) -> None:
        self.mock_deck = [
            getattr(unittest.mock.sentinel, str(x)) for x in range(52)
        ]
        self.mock_card = unittest.mock.Mock(
            side_effect=self.mock_deck)
        self.mock_rng = unittest.mock.Mock(
            wraps=random.Random())
        self.mock_rng.shuffle = unittest.mock.Mock()

    def test_Deck3_should_deal(self) -> None:
        d = Deck3(size=1, random=self.mock_rng, card_factory=self.mock_card)
        dealt = []
        for i in range(52):
            card = d.deal()
            dealt.append(card)
        self.assertEqual(dealt, self.mock_deck)

    def test_empty_deck_should_exception(self) -> None:
        d = Deck3(size=1, random=self.mock_rng, card_factory=self.mock_card)
        for i in range(52):
            card = d.deal()
        self.assertRaises(DeckEmpty, d.deal)
```

This mock for the card factory function uses the `side_effect` argument to `Mock()`. When provided with an iterable, the `side_effect` feature returns another value of the iterable each time it's called.

In this case, we used the `sentinel` object to build 52 distinct `sentinel` objects; we'll use these instead of `Card` objects to isolate the `Deck3` class from the `Card` class hierarchy. The `getattr(unittest.mock.sentinel, str(x))` expression will use a string version of a number, `x`, and create 52 unique `sentinel` objects.

We mocked the `shuffle()` method to be sure that the cards aren't actually rearranged. The wrapper means that most features of the `Random` class will be accessible. The `shuffle` method, however, was replaced. We want the `sentinel`

objects to stay in their original order so that our tests have a predictable expected value.

The first test, `test_Deck3_should_deal`, accumulates the results of dealing 52 cards into a variable, `dealt`. It then asserts that this variable has the 52 expected values from the original mock card factory. Because the card factory was a mock object, it returned the various `sentinel` objects via the `side_effect` feature of `Mock`.

The second test, `test_empty_deck_should_exception`, deals all of the cards from a `Deck` instance. However, it makes one more API request. The assertion is that the `Deck.deal()` method will raise the proper exception after dealing all of the cards.

Because of the relative simplicity of the `Deck` class, it's possible to combine both `TestDeckBuild` and `TestDeckDeal` into a single, more sophisticated mock. While that's possible with this example, it's neither essential, nor necessarily desirable to refactor the test cases to make them simpler. Indeed, the overs simplification of tests may fail to properly test API features.

Using doctest to define test cases

The `doctest` module provides us with a way to validate documentation strings. In addition to docstrings in the code, any document with a Python REPL-style response can be tested via `doctest`. This will combine the documentation for modules, classes, functions, and test cases into one tidy package.

`doctest` cases are written into docstrings. A `doctest` case shows us the interactive Python prompt, `>>>`; statements; and the expected responses. The `doctest` module contains an application that looks for these examples in docstrings. It runs the given examples and compares the expected results shown in the docstrings with the actual outputs.

With the careful design of an API, we can create a class that can be used interactively. If it can be used interactively, then a `doctest` example can be built to show the expected results of the interaction.

Indeed, two attributes of a well-designed class are that it can be used interactively and it has `doctest` examples in the documentation strings. Many built-in modules contain `doctest` examples of the API. Many other packages that we might choose to download will also include `doctest` examples.

With a simple function, we can provide documentation such as the following:

```
def ackermann(m: int, n: int) -> int:
    """Ackermann's Function
    ackermann(m, n) = $2 \\uparrow^{m-2} (n+3)-3$

    See http://en.wikipedia.org/wiki/Ackermann_function and
    http://en.wikipedia.org/wiki/Knuth%27s_up-arrow_notation.

    >>> from Chapter_17.ch17_ex1 import ackermann
    >>> ackermann(2,4)
    11
    >>> ackermann(0,4)
    5
    >>> ackermann(1,0)
    2
    >>> ackermann(1,1)
    3

    """
    if m == 0:
        return n + 1
```

```
    elif m > 0 and n == 0:
        return ackermann(m - 1, 1)
    elif m > 0 and n > 0:
        return ackermann(m - 1, ackermann(m, n - 1))
    else:
        raise LogicError()
```

We've defined a version of the Ackermann function. The function is rather complex, and the definition involves some odd-looking mathematical notation. The formal definition is shown in the following two ways:

- $\mathbf{A}(m, n) = 2 \uparrow^{m-2} (n+3) - 3$. This definition uses the extended up-arrow notation for recursive exponentiation.
- $\mathbf{A}(m, n) = \begin{cases} n+1 & \text{if } m = 0 \\ \mathbf{A}(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ \mathbf{A}(m-1, \mathbf{A}(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$. This is the recursive definition implemented previously.

The definition includes docstring comments that include five sample responses from interactive Python. The first sample output is the `import` statement, which should produce no output. The other four sample outputs show us the different values of the function.

We can run these tests with the `doctest` module. When the `doctest` module is run as a program, the command-line argument is the file that should be tested. The `doctest` program locates all docstrings and looks for interactive Python examples in those strings. It's important to note that the `doctest` documentation provides details on the regular expressions used to locate the strings. In our example, we added a hard-to-see blank line after the last `doctest` example to help the `doctest` parser.

We can run `doctest` from the command line as follows:

```
$ python3 -m doctest Chapter_17/ch17_ex1.py
```

If everything is correct, this is silent. We can make it show us some details by adding the `-v` option as follows:

```
$ python3 -m doctest -v Chapter_17/ch17_ex1.py
```

This will provide us with the details of each docstring parsed and each test case gleaned from the docstrings.

The output will include the following:

```
Trying:
 from Chapter_17.ch17_ex1 import ackermann
Expecting nothing
ok
Trying:
 ackermann(2,4)
Expecting:
 11
ok
Trying:
 ackermann(0,4)
Expecting:
 5
ok
Trying:
 ackermann(1,0)
Expecting:
 2
ok
Trying:
 ackermann(1,1)
Expecting:
 3
ok
```

The `Trying:` clause shows the statement found in a `>>>` example. `Expecting:` shows the following result lines. The final `ok` tells us the example worked as expected. The verbose output will show us all of the classes, functions, and methods without any tests, as well as the components that have tests. This provides some confirmation that our tests were properly formatted in the docstrings.

In some cases, we will have output that will not match interactive Python easily. In these cases, we may need to supplement the docstring with some annotations that modify how the test cases and expected results are parsed.

There's a special comment string that we can use for more complex outputs. We can append any one of the following two commands to enable (or disable) the various kinds of directives that are available. The following is the first command:

```
# doctest: +DIRECTIVE
```

The following is the second command:

```
# doctest: -DIRECTIVE
```

There are a dozen modifications that we can make to how the expected results

are handled. Most of them are rare situations regarding spacing and how actual and expected values should be compared.

The `doctest` documentation emphasizes the **Exact Match Principle**:

"doctest is serious about requiring exact matches in expected output."

> TIP: *If even a single character doesn't match, the test fails. You'll need to build flexibility into some of the expected outputs. If building in flexibility gets too complex, it suggests that `unitest` might be a better choice.*

Here are some specific situations where the expected and actual values of `doctest` won't match easily:

- The `id()` and default `repr()` of an object involve physical memory addresses; Python makes no guarantee that they will be consistent. If you show `id()` or `repr()`, use the `#doctest: +ELLIPSIS` directive and replace the ID or address with `...` in the sample output.
- Floating-point results may not be consistent across platforms. Always show floating-point numbers with formatting or rounding to reduce the number of digits to digits that are meaningful. Use `"{:.4f}".format(value)` or `round(value,4)` to ensure that insignificant digits are ignored.
- While dictionary keys are now ordered, set ordering is not guaranteed by Python. Use a construct such as `sorted(some_set)` instead of `some_set`.
- The current date or time, of course, cannot be used, as that won't be consistent. A test that involves date or time needs to force a specific date or time generally by mocking `time` or `datetime`.
- Operating system details, such as file sizes or timestamps, are likely to vary and should not be used without ellipses. Sometimes, it's possible to include a useful setup or teardown in the `doctest` script to manage OS resources. In other cases, mocking the `os` module is helpful.

These considerations mean that our `doctest` module may contain some additional processing that's not simply a part of the API. We may have done something such as this at the interactive Python prompt:

```
>>> sum(values)/len(values)
3.142857142857143
```

This shows us the full output from a particular implementation. We can't simply copy and paste this into a docstring. The floating-point results might differ. We'll need to do something like the following code:

```
>>> round(sum(values)/len(values), 4)
3.1429
```

This is rounded to a value that should not vary between implementations.

It is sometimes helpful to combine the `doctest` and unit tests in a comprehensive test package. We'll look at ways to include `doctest` cases with `unittest` test cases later.

Combining doctest and unittest

There's a hook in the `doctest` module that will create a proper `unittest.TestSuite` from docstring comments. This allows us to use both `doctest` and `unittest` in a large application.

What we'll do is create an instance of `doctest.DocTestSuite()`. This will build a suite from a module's docstrings. If we don't specify a module, the module that is currently running is used to build the suite. We can use a module such as the following one:

```
import doctest
suite5 = doctest.DocTestSuite()
t = unittest.TextTestRunner(verbosity=2)
t.run(suite5)
```

We built a suite, `suite5`, from the `doctest` strings in the current module. We used `unittestTextTestRunner` on this suite. As an alternative, we can combine the `doctest` suite with other `TestCase` to create a larger, more complete suite.

As our tests become more complex, we need to organize our test modules. In the next section, we'll look at the big picture of creating a `tests` folder within a Python project.

Creating a more complete test package

For larger applications, each application module can have a parallel module that includes `TestCase` for that module. This can form two parallel package structures: an `src` structure with the application module and a `test` structure with the test modules. Here are two parallel directory trees that show us the collections of modules:

```
src
    __init__.py
    __main__.py
    module1.py
    module2.py
    setup.py
tests
    __init__.py
    module1.py
    module2.py
    all.py
```

Clearly, the parallelism isn't exact. We don't usually have an automated unit test for `setup.py`. A well-designed `__main__.py` may not require a separate unit test, as it shouldn't have much code in it. We'll look at some ways to design `__main__.py` in `Chapter 18`, *Coping with the Command Line*.

We can create a top-level `test/all.py` module with a body that builds all of the tests into a single suite as follows:

```
import module1
import module2
import unittest
import doctest
all_tests = unittest.TestSuite()
for mod in module1, module2:
    all_tests.addTests(mod.suite())
    all_tests.addTests(doctest.DocTestSuite(mod))
t = unittest.TextTestRunner()
t.run(all_tests)
```

We built a single suite, `all_tests`, from the suites within the other test modules. This provides us with a handy script that will run all of the tests that are available as part of the distribution.

There are ways to use the test discovery features of the `unittest` module to do this as well. We perform package-wide testing from the command line, with something like the following code:

```
python3 -m unittest tests/*.py
```

This will use the default test discovery features of `unittest` to locate `TestCase` in the given files. This has the disadvantage of relying on shell script features rather than pure Python features. The wildcard file specification can sometimes make development more complex because incomplete modules might get tested.

Using `pytest` has some advantages over using `unittest` to discover and run the overall suite of tests. As we'll see later, `pytest` is somewhat simpler. More importantly, it can locate a wider variety of test cases, including functions as well as subclasses of `unittest.TestCase`. This allows even more flexibility and more rapid development of new software features.

Using setup and teardown

There are three levels of setup and teardown available for the `unittest` modules. These define different kinds of testing scopes: method, class, and module, as follows:

- **Test case `setUp()` and `tearDown()` methods**: These methods ensure that each individual test method within a `TestCase` class has a proper context. Often, we'll use the `setUp()` method to create the units being tested and any mock objects that are required.
- **Test case `setUpClass()` and `tearDownClass()` methods**: These methods perform a one-time setup (and teardown) of all the tests in a `TestCase` class. These methods bracket the sequence of `setUp()-testMethod()-tearDown()` for each method. This can be a good place to insert and delete the data required by the test inside a database.
- **Module `setUpModule()` and `tearDownModule()` functions**: These two standalone functions provide us with a one-time setup before all of the `TestCase` classes in a module. This is a good place to create and destroy a test database as a whole before running a series of `TestCase` classes.

We rarely need to define all of these `setUp()` and `tearDown()` methods. There are several testing scenarios that are going to be part of our design for testability. The essential difference between these scenarios is the degree of integration involved. As noted previously, we have three tiers in our testing hierarchy: isolated unit tests, integration tests, and overall application tests. There are several ways in which these tiers of testing work with the various setup and teardown features including the following:

- **Isolated unit, no integration, no dependencies.** Some classes or functions have no external dependencies; they don't rely on files, devices, other processes, or other hosts. Other classes have some external resources that can be mocked. When the cost and complexity of the `TestCase.setUp()` method is small, we can create the necessary objects there. If the mock objects are particularly complex, a class-level `TestCase.setUpClass()` might be more appropriate to amortize the cost of recreating the mock objects over several test methods.

- **Internal integration, some dependencies:** Automated integration testing among classes or modules often involves more complex setup situations. We may have a complex `setUpClass()` or even a module-level `setUpModule()` to prepare a context for integration testing. When working with the database access layers in Chapter 11, *Storing and Retrieving Objects via Shelve*, and Chapter 12, *Storing and Retrieving Objects via SQLite*, we often perform integration testing that includes our class definitions as well as our access layer. This may involve seeding a test database or shelf with appropriate data for the tests.
- **External integration**: We may perform automated integration testing on larger and more complex pieces of an application. In these cases, we may need to spawn external processes or create databases and seed them with data. In this case, we may have `setUpModule()` to prepare an empty database for use by all of the `TestCase` classes in a module. When working with RESTful web services in Chapter 13, *Transmitting and Sharing Objects*, or testing **Programming In The Large** (**PITL**) in Chapter 19, *Module and Package Design*, this approach could be helpful.

Note that the concept of unit testing does not define what the unit under test is. The *unit* can be a class, a module, a package, or even an integrated collection of software components. The unit needs to be isolated from its environment to create a focused test.

When designing automated integration tests, it's important to choose the components to be tested. For example, we don't need to test Python libraries; they have their own tests. Similarly, we don't need to test the OS. For example, if our software deletes a file, we don't need to include a check of the filesystem integrity after the file is removed. We don't need to be sure the space was reclaimed. We generally need to trust that libraries and operating systems work correctly. If we're doubtful of the infrastructure, we can run the test suite for the OS or libraries; we don't need to reinvent it. An integration test must focus on testing the code we wrote, not the code we downloaded and installed.

Using setup and teardown with OS resources

In many cases, a test case may require a particular OS environment. When working with external resources such as files, directories, or processes, we may need to create or initialize them before a test. We may also need to remove the resources before a test, or we may need to tear down the resources at the end of the test.

Let's assume that we have a function, `rounds_final()`, which is supposed to process a given file. We need to test the function's behavior in the rare case that the file doesn't exist. It's common to see `TestCase` with a structure such as the following one:

```
import os

class Test_Missing(unittest.TestCase):

    def setUp(self) -> None:
        try:
            (Path.cwd() / "data" / "ch17_sample.csv").unlink()
        except OSError as e:
            pass  # OK if file didn't exist

    def test_missingFile_should_returnDefault(self) -> None:
        self.assertRaises(
            FileNotFoundError, rounds_final, (Path.cwd() / "data" / "ch17_sample.csv")
        )
```

We have to handle the possible exception of trying to remove a file that doesn't exist in the first place. This test case has a `setUp()` method that ensures that the required file is missing. Once `setuUp()`ensures that the file is truly gone, we can execute the `rounds_final()` function with an argument of the missing file, `p3_c15_sample.csv`. We expect this to raise a `FileNotFoundError` error.

Note that raising `FileNotFoundError` is a default behavior of Python's `open()` method. This may not require testing at all. This leads to an important question: *why test a built-in feature*? If we're performing black box testing, we need to exercise all features of the external interface, including the expected default behaviors. If we're performing white box testing, we may need to test the exception handling

`try:` statement within the body of the `rounds_final()` function.

The `ch17_sample.csv` filename is repeated within the body of the test. Some folks feel that the DRY rule should apply even to test code. There's a limit to how much of this kind of optimization is valuable while writing tests:

> TIP *It's okay for test code to be brittle. If a small change to the application leads to test failures, this really is a good thing. Tests should value simplicity and clarity, not robustness and reliability.*

In the next section, we'll look at some other examples of using `setUp()` and `tearDown()` for databases.

Using setup and teardown with databases

When working with a database and ORM layer, we often have to create test databases, files, directories, or server processes. We may need to tear down a test database after the tests pass, to be sure that the other tests can run. We may not want to tear down a database after failed tests; we may need to leave the database alone so that we can examine the resulting rows to diagnose the test failures.

It's important to manage the scope of testing in a complex, multilayered architecture. Looking back at `Chapter 12`, *Storing and Retrieving Objects via SQLite*, we don't need to specifically test the SQLAlchemy ORM layer or the SQLite database. These components have their own test procedures outside our application tests. However, because of the way the ORM layer creates database definitions, SQL statements, and Python objects from our code, we can't easily mock SQLAlchemy and hope that we've used it properly. We need to test the way our application uses the ORM layer without digressing into testing the ORM layer itself.

One of the more complex test case setup situations will involve creating a database and then populating it with appropriate sample data for the given test. When working with SQL, this can involve running a fairly complex script of SQL DDL to create the necessary tables and then another script of SQL DML to populate those tables. The associated teardown will be another complex SQL DDL script.

This kind of test case can become long-winded, so we'll break it into three sections: a useful function to create a database and schema, the `setUpClass()` method, and the rest of the unit test.

Here's the create database function:

```
from Chapter_12.ch12_ex4 import Base, Blog, Post, Tag, assoc_post_tag
import datetime
import sqlalchemy.exc
```

```
from sqlalchemy import create_engine

def build_test_db(name="sqlite:///./data/ch17_blog.db"):
    """
    Create Test Database and Schema
    """
    engine = create_engine(name, echo=True)
    Base.metadata.drop_all(engine)
    Base.metadata.create_all(engine)
    return engine
```

This builds a fresh database by dropping all of the tables associated with the ORM classes and recreating the tables. The idea is to ensure a fresh, empty database that conforms to the current design, no matter how much that design has changed since the last time the unit tests were run.

In this example, we built an SQLite database using a file. We can use the *in-memory* SQLite database feature to make the test run somewhat more quickly. The downside of using an in-memory database is that we have no database that we can use to debug failed tests.

Here's how we use this in a `TestCase` subclass:

```
from sqlalchemy.orm import sessionmaker


class Test_Blog_Queries(unittest.TestCase):

    @staticmethod
    def setUpClass():
        engine = build_test_db()
        Test_Blog_Queries.Session = sessionmaker(bind=engine)
        session = Test_Blog_Queries.Session()

        tag_rr = Tag(phrase="#RedRanger")
        session.add(tag_rr)
        tag_w42 = Tag(phrase="#Whitby42")
        session.add(tag_w42)
        tag_icw = Tag(phrase="#ICW")
        session.add(tag_icw)
        tag_mis = Tag(phrase="#Mistakes")
        session.add(tag_mis)

        blog1 = Blog(title="Travel 2013")
        session.add(blog1)
        b1p1 = Post(
            date=datetime.datetime(2013, 11, 14, 17, 25),
            title="Hard Aground",
            rst_text="""Some embarrassing revelation. Including ☹ and ⚓""",
            blog=blog1,
            tags=[tag_rr, tag_w42, tag_icw],
        )
        session.add(b1p1)
        b1p2 = Post(
            date=datetime.datetime(2013, 11, 18, 15, 30),
            title="Anchor Follies",
```

```
            rst_text="""Some witty epigram. Including ☺ and ☀""",
            blog=blog1,
            tags=[tag_rr, tag_w42, tag_mis],
        )
        session.add(b1p2)

        blog2 = Blog(title="Travel 2014")
        session.add(blog2)
        session.commit()
```

We defined `setUpClass()` so that a database is created before the tests from this class are run. This allows us to define a number of test methods that will share a common database configuration. Once the database has been built, we can create a session and add data.

We've put the session maker object into the class as a class-level attribute, `Test_Blog_Queries.Session = sessionmaker(bind=engine)`. This class-level object can then be used in `setUp()` and individual test methods.

Here is `setUp()` and two of the individual test methods:

```
def setUp(self):
    self.session = Test_Blog_Queries.Session()

def test_query_eqTitle_should_return1Blog(self):
    results = self.session.query(Blog).filter(Blog.title == "Travel 2013").all()
    self.assertEqual(1, len(results))
    self.assertEqual(2, len(results[0].entries))

def test_query_likeTitle_should_return2Blog(self):
    results = self.session.query(Blog).filter(Blog.title.like("Travel %")).all()
    self.assertEqual(2, len(results))
```

The `setUp()` method creates a new, empty session object from the class-level `sessionmaker` instance. This will ensure that every query is able to properly generate SQL and fetch data from the database using an SQLAlchemy session.

The `query_eqTitle_should_return1Blog()` test will find the requested `Blog` instance and navigate to the `Post` instances via the `entries` relationship. The `filter()` portion of the request doesn't really test our application definitions; it exercises SQLAlchemy and SQLite. The `results[0].entries` test in the final assertion is a meaningful test of our class definitions.

The `query_likeTitle_should_return2Blog()` test is almost entirely a test of SQLAlchemy and SQLite. It isn't really making meaningful use of anything in our application except the presence of an attribute named `title` in `Blog`. These kinds of tests are often left over from creating initial technical spikes. They can

help clarify an application API, even if they don't provide much value as a test case.

Here are two more test methods:

```
def test_query_eqW42_tag_should_return2Post(self):
    results = self.session.query(Post).join(assoc_post_tag).join(Tag).filter(
        Tag.phrase == "#Whitby42"
    ).all()
    self.assertEqual(2, len(results))

def test_query_eqICW_tag_should_return1Post(self):
    results = self.session.query(Post).join(assoc_post_tag).join(Tag).filter(
        Tag.phrase == "#ICW"
    ).all()
    self.assertEqual(1, len(results))
    self.assertEqual("Hard Aground", results[0].title)
    self.assertEqual("Travel 2013", results[0].blog.title)
    self.assertEqual(
        set(["#RedRanger", "#Whitby42", "#ICW"]),
        set(t.phrase for t in results[0].tags),
    )
```

The `query_eqW42_tag_should_return2Post()` test performs a more complex query to locate the posts that have a given tag. This exercises a number of relationships defined in the classes. When both of the relevant blog entries are located, this test has been passed.

The `query_eqICW_tag_should_return1Post()` test, similarly, exercises a complex query. It tests the navigation from `Post` to the `Blog` instance which contains the `Post` via `results[0].blog.title`. It also tests navigation from `Post` to an associated collection of `Tags` via `set(t.phrase for t in results[0].tags)`. We must use an explicit `set()` because the order of results in SQL is not guaranteed.

What's important about this `Test_Blog_Queries` subclass of `TestCase` is that it creates a database schema and a specific set of defined rows via the `setUpClass()` method. This kind of test setup is helpful for database applications. It can become rather complex and is often supplemented by loading sample rows from files or JSON documents, rather than coding the rows in Python.

The TestCase class hierarchy

Inheritance works among the `TestCase` classes. Ideally, each `TestCase` is unique. Pragmatically, there may be common features among cases. There are the following three common ways in which `TestCase` classes may overlap:

- **Common** `setUp()`: We may have some data that is used in multiple `TestCase`. There's no reason to repeat the data. A `TestCase` class that only defines `setUp()` or `tearDown()` with no test methods is legal, but it may lead to a confusing log, because there are zero tests involved.
- **Common** `tearDown()`: It's common to have a common cleanup for tests that involve OS resources. We might need to remove files and directories or kill subprocesses.
- **Common results checking**: For algorithmically complex tests, we may have a results checking method that verifies some properties of a result.

Looking back at `Chapter 4`, *Attribute Access, Properties, and Descriptors*, for example, consider the `RateTimeDistance` class. This class fills in a missing value in a dictionary based on two other values, as follows:

```
@dataclass
class RateTimeDistance:

    rate: Optional[float] = None
    time: Optional[float] = None
    distance: Optional[float] = None

    def __post_init__(self) -> None:
        if self.rate is not None and self.time is not None:
            self.distance = self.rate * self.time
        elif self.rate is not None and self.distance is not None:
            self.time = self.distance / self.rate
        elif self.time is not None and self.distance is not None:
            self.rate = self.distance / self.time
```

Each unit test method for the `RateTimeDistance` class can include the following code:

```
self.assertAlmostEqual(
    self.rtd.distance, self.rtd.rate * self.rtd.time, places=2
)
```

If we use a number of `TestCase` subclasses, we can inherit this validity check as a

separate method as follows:

```
def validate(self, object):
    self.assertAlmostEqual(
        self.rtd.distance, self.rtd.rate * self.rtd.time, places=2
    )
```

This way, each test need only include `self.validate(object)` to be sure that all the tests provide a consistent definition of correctness.

An important feature of the definition of the `unittest` module is that the test cases are proper classes with proper inheritance. We can design the `TestCase` class hierarchy with the same care and attention to detail that we apply to the application classes.

Using externally defined expected results

For some applications, the users can articulate processing rules that describe the software's behavior. In other cases, the job of an analyst or a designer transforms the user's desires into procedural descriptions of the software.

It's often easiest to provide concrete examples of expected results. Either the ultimate users or intermediary analysts may find it helpful to create a spreadsheet that shows sample inputs and expected results. Working from user-supplied, concrete sample data can simplify the software being developed.

Whenever possible, have real users produce concrete examples of correct results. Creating procedural descriptions or software specifications is remarkably difficult. Creating concrete examples and generalizing from the examples to a software specification is less fraught with complexity and confusion. Further, it plays into a style of development where test cases drive development efforts. A suite of test cases provides a developer with a concrete definition of **done**. Tracking the software development project status leads to asking how many test cases we have today and how many of them pass.

Given a spreadsheet of concrete examples, we need to turn each row into a `TestCase` instance. We can then build a suite from these objects.

For the previous examples in this chapter, we loaded the test cases from a `TestCase`-based class. We used `unittest.defaultTestLoader.loadTestsFromTestCase` to locate all the methods with a name that starts with `test`. The loader creates a test object from each method with the proper name prefix and combines them into a test suite.

There's an alternative approach, however. For this example, we're going to build test case instances individually. This is done by defining a class with a single `runTest()` method. We can load multiple instances of this class into a suite. For this to work, the `TestCase` class must define only one test with the name `runTest()`. We won't be using the loader to create the test objects; we'll be creating them

directly from rows of externally supplied data.

Let's take a look at a concrete function that we need to test. This is from `Chapter 4`, *Attribute Access, Properties, and Descriptors*:

```
from Chapter_4.ch04_ex3 import RateTimeDistance
```

This is a class that eagerly computes a number of attributes when it is initialized. The users of this simple function provided us with some test cases as a spreadsheet, from which we extracted the CSV file. For more information on CSV files, see `Chapter 10`, *Serializing and Saving - JSON, YAML, Pickle, CSV, and XML*. We need to transform each row into a `TestCase`. Here's the data in the CSV file:

```
rate_in,time_in,distance_in,rate_out,time_out,distance_out
2,3,,2,3,6
5,,7,5,1.4,7
,11,13,1.18,11,13
```

We're not going to define a class using a name that starts with `test` because the class isn't going to be simply discovered by a loader. Instead, the class is used to build instances into a larger suit of tests. Here's the test case template that we can use to create test instances from each row of the CSV file:

```
class Test_RTD(unittest.TestCase):

    def runTest(self) -> None:
        with (Path.cwd() / "data" / "ch17_data.csv").open() as source:
            rdr = csv.DictReader(source)
            for row in rdr:
                self.example(**row)

    def example(
        self,
        rate_in: str,
        time_in: str,
        distance_in: str,
        rate_out: str,
        time_out: str,
        distance_out: str,
    ) -> None:
        args = dict(
            rate=float_or_none(rate_in),
            time=float_or_none(time_in),
            distance=float_or_none(distance_in),
        )
        expected = dict(
            rate=float(rate_out), time=float(time_out), distance=float(distance_out)
        )
        rtd = RateTimeDistance(**args)
        assert rtd.distance and rtd.rate and rtd.time
        self.assertAlmostEqual(rtd.distance, rtd.rate * rtd.time, places=2)
```

```
        self.assertAlmostEqual(rtd.rate, expected["rate"], places=2)
        self.assertAlmostEqual(rtd.time, expected["time"], places=2)
        self.assertAlmostEqual(rtd.distance, expected["distance"], places=2)
```

The testing is embodied in the `runTest()` method of this class. In previous examples, we used method names starting with `test_` to provide the test case behavior. Instead of multiple `test_` method names, a single `runTest()` method can be provided. This will also change the way a test suite is built, as we'll see next.

This method parses a row of a spreadsheet into a dictionary. For this to work correctly, the sample data column headings must match the parameter names required by the `example()` method. The input values are placed in a dictionary named `args`; the expected result values are, similarly, placed into a dictionary named `expected`.

The `float_or_none()` function helps handle the CSV source data where a `None` value will be represented by an empty string. It converts the text of a cell to a `float` value or `None`. The function is defined as follows:

```
def float_or_none(text: str) -> Optional[float]:
    if len(text) == 0:
        return None
    return float(text)
```

Each row of the spreadsheet is processed through the `example()` method. This gives us a relatively flexible approach to testing. We can permit users or business analysts to create all the examples required to clarify proper operation.

We can build a suite from this test object as follows:

```
def suite9():
    suite = unittest.TestSuite()
    suite.addTest(Test_RTD())
    return suite
```

Note that we do not use the `loadTestsFromTestCase` method to discover the methods with `test_` names. Instead, we create an instance of the test case that can simply be added to the test suite.

The suite is executed using the kind of script we've seen earlier. Here's an example:

```
if __name__ == "__main__":
    t = unittest.TextTestRunner()
    t.run(suite9())
```

The output looks like this:

```
..F
======================================================================
FAIL: runTest (__main__.Test_RTD)
{'rate': None, 'distance': 13.0, 'time': 11.0} -> {'rate': 1.18, 'distance': 13.0, 'time
----------------------------------------------------------------------
Traceback (most recent call last):
  File "p3_c15.py", line 504, in runTest
    self.assertAlmostEqual( self.rtd.rate, self.result['rate'] )
AssertionError: 1.1818181818181819 != 1.18 within 7 places

----------------------------------------------------------------------
Ran 3 tests in 0.000s

FAILED (failures=1)
```

The user-supplied data has a small problem. The users provided a value that has been rounded off to only two places. Either the sample data needs to provide more digits, or our test assertions need to cope with the rounding.

This can also be run from the command line using `unittest` test discovery. Here's the command we can run to use the built-in test discovery features of the `unittest` module:

```
python3 -m unittest Chapter_17/ch17_ex1.py
```

This produces an abbreviated output with many of the testing examples from this chapter. It looks like this:

```
.x...........x..Run time 0.931446542
..
----------------------------------------------------------------------
Ran 19 tests in 0.939s
```

Each `.` is a test that passed. The `x` marks are tests that are expected to fail. As we noted previously, some of the tests reveal problems with the defined classes, and those tests will fail until the classes are fixed.

The `Run time 0.931446542` output is from a `print()` within a test. It's not a standard part of the output. Because of the way the output is structured, it can be difficult to print other debugging or performance data output inside a test case. This example shows how it can be confusing to have the additional output in the middle of the simple line of periods showing test execution progress.

Using pytest and fixtures

An alternative to the `unittest` runner is the `pytest` test runner. The `pytest` framework has an excellent test discovery feature that goes beyond what the `unittest` tool can discover.

The `unittest` runner can be used in the following two ways.

- With a test suite object. The previous examples have focused on this.
- To search for classes that are extensions of `unittest.TestCase`, build a test suite from those, and then run the suite. This offers considerable flexibility. We can add test cases without also having to update the code to build the test suite.

The `pytest` tool can also locate `unittest.TestCase` class definitions, build a suite of tests, and execute the tests. It can go beyond this and also locate functions with names starting with `test_` in modules where the name starts with `test_`. Using simple functions has some advantages over the more complex `unittest.TestCase` class definitions.

The primary advantage of using separate functions is the resulting simplification of the test module. In particular, when we look back at the `TestCardFactory` example, we see that there is no `setUp()` required for the tests within the class. Because all of the methods are independent, there's no real need to bind these methods into a single class definition. Even though this is a book on object-oriented Python, there's no reason to use class definitions when they don't improve the code. In many cases, class-oriented test case definition isn't helpful and separate functions executed by `pytest` have advantages.

The `pytest` approach leads to the following two other consequences:

- The `self.assert...()` methods are not available. When using `pytest`, the Python `assert` statement is used to compare expected results with actual results.
- The stateful class variables used by `setUp()` and `tearDown()` aren't available. In order to set up and tear down test contexts, we'll use the `pytest` `@fixture`

functions.

As a concrete example of the simplifications possible, we'll review some examples from earlier, starting with tests for the `Card` class. The `pytest` version is as follows:

```
def test_card():
    three_clubs = Card(3, Suit.CLUB)
    assert "3♣" == str(three_clubs)
    assert 3 == three_clubs.rank
    assert Suit.CLUB == three_clubs.suit
    assert 3 == three_clubs.hard
    assert 3 == three_clubs.soft
```

The function's name must begin with `test_` to be sure `pytest` can discover it. The test setup creates a `card` instance, and a number of `assert` statements to confirm that it has the expected behavior. We don't have quite as much overhead when using functions with the `assert` statement as we do when using `unitest.TestCase` subclasses.

Assertion checking

To confirm the exception raised by a test, the `unittest.TestCase` class has methods like `assertRaises()`. When working with `pytest`, we have a distinct approach to testing this feature. The `pytest` package offers a context manager named `raises` to help detect the exception that is raised. The `raises` context is shown in this example:

```
from pytest import raises

def test_card_factory():

    c1 = card(1, Suit.CLUB)
    assert isinstance(c1, AceCard)

    c2 = card(2, Suit.DIAMOND)
    assert isinstance(c1, Card)

    c10 = card(10, Suit.HEART)
    assert isinstance(c10, Card)

    cj = card(11, Suit.SPADE)
    assert isinstance(cj, FaceCard)

    ck = card(13, Suit.CLUB)
    assert isinstance(ck, FaceCard)

    with raises(LogicError):
        c14 = card(14, Suit.DIAMOND)

    with raises(LogicError):
        c0 = card(0, Suit.DIAMOND)
```

We've used `pytest.raises` as a context manager. When this is provided with a class definition, the statements within the context are expected to raise the named exception. If the exception is raised, the test passes; if the exception is not raised, this is a test failure.

Using fixtures for test setup

The test setup and teardown features of `pytest` are often handled by `@fixture` functions. These functions form the fixture into which a unit is connected for testing. In the realm of hardware testing, it might also be called a test harness or test bench.

Fixtures can be used to do any kind of setup or teardown related to a test. Because of the way `pytest` invokes a test, it implicitly calls the fixture functions, simplifying our test code considerably. The fixtures can reference other fixtures, letting us create composite objects that can help to isolate the unit being tested.

Previously, we looked at two complex test case subclasses: `TestDeckBuild` and `TestDeckDeal`. These two test cases covered separate features of the `Deck3` class definition. We can build similar test cases using a common fixture. Here's the fixture definition:

```
import unittest.mock
from types import SimpleNamespace
from pytest import fixture

@fixture
def deck_context():
    mock_deck = [
        getattr(unittest.mock.sentinel, str(x))
        for x in range(52)
    ]
    mock_card = unittest.mock.Mock(side_effect=mock_deck)
    mock_rng = unittest.mock.Mock(
        wraps=random.Random,
        shuffle=unittest.mock.Mock(return_value=None)
    )
    return SimpleNamespace(**locals())
```

The `deck_context()` function, creates the following three mock objects:

- The `mock_deck` object is a list of 52 individual `mock.sentinel` objects. Each `sentinel` object is customized by getting a unique attribute of the `sentinel`. The attribute name is a string built from the integer values in `range(52)`. There will be objects with names such as `mock.sentinel.0`. This is not a valid syntax for a simple Python attribute reference in source code, but we only need to be sure the `sentinel` is unique.
- The `mock_card` object is a mock with a `side_effect`. This will behave like a

function. Each time it's invoked, it will return another value from the list provided to the `side_effect` parameter. This can be used to simulate a function that reads values from files or a network connection.

- The `mock_rng` object is a wrapped version of the `random.Random` class. This will behave like a random object, except for two features. First, the `shuffle()` method doesn't do anything. And, second, the `Mock` wrapper will track individual calls to the methods of this object so we can determine whether it's being used properly by the unit being tested.

The `return` step packages all of the local variables into a single `SimpleNamespace` object. This object lets us use syntax such as `deck_context.mock_card` to refer to the `Mock` function. We can use this fixture in a `test` function. The following is an example:

```
def test_deck_build(deck_context):
    d = Deck3(
        size=1,
        random=deck_context.mock_rng,
        card_factory=deck_context.mock_card
    )
    deck_context.mock_rng.shuffle.assert_called_once_with(d)
    assert 52 == len(deck_context.mock_card.mock_calls)
    expected = [
        unittest.mock.call(r, s) for r in range(1, 14) for s in iter(Suit)
    ]
    assert expected == deck_context.mock_card.mock_calls
```

This test references the `deck_context` fixture. Nothing special is done in the code; `pytest` will implicitly evaluate the function, and the resulting `SimpleNamespace` object will be the value of the `deck_context` parameter. The mapping between the parameter and the fixture is very simple: all parameter names must be the names of fixture functions, and these functions are evaluated automatically.

The test builds a `Deck3` instance using mock objects for the `random` parameter and the `card_factory` parameter. Once the `Deck3` instance is built, we can examine the mock objects to see if they had the proper number of calls with the expected argument values.

Using fixtures for setup and teardown

The fixture functions can also provide a teardown capability as well as a setup capability. This relies on the lazy way generator functions work. Some example code for a fixture that does setup and teardown is as follows:

```
@fixture
def damaged_file_path():
    file_path = Path.cwd() / "data" / "ch17_sample.csv"
    with file_path.open("w", newline="") as target:
        print("not_player,bet,rounds,final", file=target)
        print("data,1,1,1", file=target)
    yield file_path
    file_path.unlink()
```

The `damaged_file_path()` function creates a file with a relative path of `data/ch17_sample.csv`. A few lines of data are written to the file.

The `yield` statement provides an initial result value. This is used by a test function. When the test completes, then a second value is retrieved from the fixture. When this second result is requested, the fixture function can do any teardown work that is required. In this example, the teardown work deletes the test file that was created.

The fixture function will be called implicitly when the test is run. Here is an example test that uses this fixture:

```
def test_damaged(damaged_file_path):
    with raises(AssertionError):
        stats = rounds_final(Path.cwd()/"data"/"ch17_sample.csv")
```

This test confirms that the `rounds_final()` function, when given the example file, will raise `AssertionError`. Because the `damaged_file_path` fixture uses `yield`, it can tear down the test context, removing the file.

Building parameterized fixtures

It's common to have a large number of similar examples for a test case. In previous sections, we talked about having users or analysts create spreadsheets with examples of inputs and outputs. This can be helpful for permitting direct input to the software development process. We need our testing tools to work with the CSV example files as directly as possible.

With `pytest`, we can apply parameters to a fixture. The `pytest` runner will use each object in the parameter collection to run the test function repeatedly. To build a parameterized fixture, we can use code like the following example:

```
import csv

with (Path.cwd() / "Chapter_17" / "ch17_data.csv").open() as source:
    rdr = csv.DictReader(source)
    rtd_cases = list(rdr)

@fixture(params=rtd_cases)
def rtd_example(request):
    yield request.param
```

We've opened the CSV file in a context manager. The file is used to build a reader that transforms each row of data into a dictionary. The keys are the column titles and the values are the strings from each cell in a given row. The final `rtd_cases` variable will be a list of dictionaries; a type hint of `List[Dict[str, str]]` would capture the structure.

The `rtd_example` fixture was built with the `params=` argument. Each item in the `params` collection will be provided as a context to a function under test. This means the test will be run several times, and each time it will use a different value from the `params` sequence. To use this fixture, we'll have a test case like the following example:

```
from pytest import approx

def test_rtd(rtd_example):
    args = dict(
        rate=float_or_none(rtd_example['rate_in']),
        time=float_or_none(rtd_example['time_in']),
        distance=float_or_none(rtd_example['distance_in']),
    )
    result = dict(
        rate=float_or_none(rtd_example['rate_out']),
```

```
        time=float_or_none(rtd_example['time_out']),
        distance=float_or_none(rtd_example['distance_out']),
    )

    rtd = RateTimeDistance(**args)

    assert rtd.distance == approx(rtd.rate * rtd.time)
    assert rtd.rate == approx(result["rate"], abs=1E-2)
    assert rtd.time == approx(result["time"])
    assert rtd.distance == approx(result["distance"])
```

This case depends on the `rtd_example` fixture. Since the fixture had a list of values for the params, this case will be called multiple times; each time, the value of `rtd_example` will be a different row from the sequence of values. This makes it convenient to write a common test for a variety of input values.

This test also uses the `pytest.approx` object. This object is used to wrap floating-point values so the `__eq__()` method is an approximately equal algorithm instead of a simple exact equality test. This is a very convenient way to ignore the tiny floating-point discrepancies that arise from the truncation of the binary representation of a number.

Automated integration or performance testing

We can use the `unittest` package to perform testing that isn't focused on a single, isolated class definition. As noted previously, we can use `unittest` automation to test a unit that is an integration of multiple components. This kind of testing can only be performed on software that has passed unit tests on isolated components. There's no point in trying to debug a failed integration test when a component's unit test didn't work correctly.

Performance testing can be done at several levels of integration. For a large application, performance testing with the entire build may not be completely helpful. One traditional view is that a program spends 90 percent of its time executing just 10 percent of the available code. Therefore, we don't often need to optimize an entire application; we only need to locate the small fraction of the program that represents the real performance bottleneck.

In some cases, it will be clear that we have a data structure that involves searching. We know that removing searching will lead to a tremendous improvement in performance. As we saw in `Chapter 6`, *Using Callables and Contexts*, implementing memoization can lead to dramatic performance improvements by avoiding recalculation.

In order to perform proper performance testing, we need to follow this three-step work cycle:

1. Use a combination of design reviews and code profiling to locate the parts of the application that are likely to be a performance problem. Python has two profiling modules in the standard library. Unless there are more complex requirements, `cProfile` will locate the part of the application that requires focus.
2. Create an automated test scenario with `unittest` to demonstrate any actual performance problems. Collect the performance data with `timeit` or `time.perf_counter()`.
3. Optimize the code for the selected test case until the performance is

acceptable.

The point is to automate as much as possible and avoid vaguely tweaking things in the hope of an improvement in performance. Most of the time, a central data structure or algorithm (or both) must be replaced, leading to extensive refactoring. Having automated unit tests makes wholesale refactoring practical.

An awkward situation can arise when a performance test lacks specific pass-fail criteria. There can be a drive to make software *faster* without a concrete definition of *fast enough*. It's always simpler when there are measurable performance objectives. Given a concrete objective, then formal, automated testing can be used to assert both that the results are correct and that the time taken to get those results is acceptable.

For performance testing, we might use something like the following code:

```
import unittest
import timeit


class Test_Performance(unittest.TestCase):

    def test_simpleCalc_shouldbe_fastEnough(self):
        t = timeit.timeit(
            stmt="""RateTimeDistance(rate=1, time=2)""",
            setup="""from Chapter_4.ch04_ex3 import RateTimeDistance""",
        )
        print("Run time", t)
        self.assertLess(t, 10, f"run time {t} >= 10")
```

This use of `unittest` can create an automated performance test. As the `timeit` module executes the given statement 1,000,000 times, this should minimize the variability in the measurements from the background work on the computer that does the testing.

In the preceding example, each execution of the RTD constructor is required to take less than 1/100,000 of a second. A million executions should take less than 10 seconds.

Summary

We looked at using `unittest` and `doctest` to create automated unit tests. We also looked at creating a test suite so that collections of tests can be packaged for reuse and aggregation into suites with larger scopes, without relying on the automated test discovery process.

We looked at how to create mock objects so that we can test software units in isolation. We also looked at the various kinds of setup and teardown features. These allowed us to write tests with complex initial states or persistent results.

The FIRST unit test properties fit well with both `doctest` and `unittest`. The FIRST properties are as follows:

- **Fast**: Unless we write egregiously bad tests, the performance of `doctest` and `unitest` should be very fast.
- **Isolated**: The `unittest` package offers us a mock module that we can use to isolate our class definitions. In addition, we can exercise some care in our design to ensure that our components are isolated from each other.
- **Repeatable**: Using `doctest` and `unittest` for automated testing ensures repeatability.
- **Self-validating**: Both `doctest` and `unittest` bind test results with the test case condition, ensuring that no subjective judgment is involved in testing.
- **Timely**: We can write and run test cases as soon as we have the skeleton of a class, function, or module. A class whose body has simply `pass` is sufficient to run the test script.

For the purposes of project management, a count of written tests and passed tests is sometimes a very useful status report.

Design considerations and trade-offs

Test cases are required to be deliverable when creating software. Any feature that is without an automated test might as well not exist. A feature certainly can't be trusted to be correct if there's no test. If it can't be trusted, it shouldn't be used.

The only real trade-off question is whether to use `doctest` or `unittest` or both. For simple programming, `doctest` may be perfectly suitable. For more complex situations, `unittest` will be necessary. For frameworks where the API documentation needs to include examples, a combination works well.

In some cases, simply creating a module full of `TestCase` class definitions may be sufficient. The `TestLoader` class and test discovery features may be perfectly adequate to locate all of the tests.

More generally, `unittest` involves using `TestLoader` to extract multiple test methods from each `TestCase` subclass. We package the test methods into a single class based on who they can share class-level `setUp()`, and possibly `setUpClass()`, methods with.

We can also create `TestCase` instances without `TestLoader`. In this case, the default method of `runTest()` is defined to have the test case assertions. We can create a suite from instances of this kind of class.

The most difficult part can be designing for testability. Removing dependencies so that units can be tested independently can sometimes feel like adding to the software design's complexity. In most cases, the time expended exposing dependencies is time invested in creating more maintainable and more flexible software.

The general rule is this: *an implicit dependency between classes is bad design*.

A testable design has explicit dependencies; these can easily be replaced with mock objects. The `pytest` framework provides the `monkeypatch` fixture. This permits us to write tests that isolate the unit being tested by patching the dependencies. While this is handy, it's often simpler and more reliable to provide for simple,

visible dependency injection.

Looking forward

In the `Chapter 18`, *Coping With the Command Line*, we will look at writing complete applications that are started from the command line. We'll look at ways to handle startup options, environment variables, and configuration files in Python applications.

In `Chapter 19`, *Module and Package Design*, we'll expand on the application design. We'll add the ability to compose applications in larger applications, as well as to decompose applications into smaller modules or packages.

Coping with the Command Line

Command-line startup options, environment variables, and configuration files are important to many applications, particularly when it comes to the implementation of servers. There are a number of ways of dealing with program startup and object creation. This chapter will focus on argument parsing, and the overall architecture of an application.

This chapter will extend the configuration file handling from `Chapter 14`, *Configuration Files and Persistence*, with yet more techniques for command-line programs and the top level of a server. The core design principles from `Chapter 15`, *Design Principles and Patterns*, are essential when designing an application of any size. This chapter will also extend some logging design features from `Chapter 16`, *The Logging and Warning Modules*.

In `Chapter 19`, *Module and Package Design*, we'll extend these principles to look at a kind of architectural design that we'll call *programming in the large*. We'll use the Command design pattern to define software components that can be aggregated without resorting to shell scripts. This is particularly helpful when writing the background processing components used by application servers.

Technical requirements

The code files for this chapter are available at `https://git.io/fj2UD`.

The OS interface and the command line

Generally, the operating system's shell starts applications with several pieces of information that constitute the OS API:

- The shell provides each application with its collection of environment variables. In Python, these are accessed through `os.environ`.
- The shell prepares three standard files. In Python, these are mapped to `sys.stdin`, `sys.stdout`, and `sys.stderr`. There are some other modules, such as `fileinput`, that can provide access to `sys.stdin`.
- The command line is parsed by the shell into words. Parts of the command line are available in `sys.argv`. For POSIX operating systems, the shell may replace shell environment variables and glob wildcard filenames. In Windows, the simple `cmd.exe` shell will not glob filenames for us.
- The OS also maintains context settings, such as the current working directory, user identity, and user group information, among many other things. These are available through the `os` module. They aren't provided as arguments on the command line.

The OS expects an application to provide a numeric status code when it terminates. If we want to return a specific numeric code, we can use `sys.exit()` in our applications. The `os` module defines a number of values, such as `os.EX_OK`, to help return codes with common meanings. Python will return a zero if our program is terminated normally, a value of one if the program ended with an unhandled exception, and a value of two if the command-line arguments were invalid.

The shell's operation is an important part of this OS API. Given a line of input, the shell performs a number of substitutions, depending on the (rather complex) quoting rules and substitution options. It then parses the resulting line into space-delimited words. The first word must be either a built-in shell command (such as `cd` or `set`) or it must be the name of a file, such as `python3`. The shell searches its defined `PATH` for this file.

To make effective use of executable files, it's imperative that you are sure that the directory with those files is named by the `PATH` environment variable. In most OSes, you should append a colon (`:`) and the directory for your script. In Windows, you should append a semicolon (`;`) and the directory for your script.

The file named on the first word of a command must have execute, `x`, permission. The `chmod +x somefile.py` shell command marks a file as executable. A filename that isn't executable gets an `OS Permission Denied` error. Use the OS `ls -l` (or the Windows equivalent) command to see file permissions.

The first bytes of an executable file have a magic number that is used by the shell to decide how to execute that file. Some magic numbers indicate that the file is a binary executable; the shell can fork a subshell and execute it. Other magic numbers, specifically the value encoded by two bytes `b'#!'`, indicate that the file is a proper text script and requires an interpreter. The rest of the first line of this kind of file is the name of the interpreter.

We often use a line like the following in a Python file:

```
#!/usr/bin/env python3
```

If the Python file has permission to execute, and has this as the first line, then the shell will run the `env` program. The `env` program's argument (`python3`) will cause it to set up an environment and run the Python 3 program with the Python file as the first positional argument.

After setting the `PATH` correctly, what happens when we enter `ch18_demo.py -s someinput.csv` at the command line? The sequence of steps that the program works through from the OS shell via an executable script to Python looks like the following:

1. The shell parses the `ch18_demo.py -s someinput.csv` line. The first word is `ch18_demo.py`. This file is on the shell's `PATH` and has the `x` executable permission. The shell opens the file and finds the `#!` bytes. The shell reads the rest of this line and finds the `/usr/bin/env python3` command.

2. The shell parses the new `/usr/bin/env` command, which is a binary executable. The shell starts the `env` program. This program, in turn, starts `python3`. The sequence of words from the original command line, as parsed

by the shell `['ch18_demo.py', '-s', 'someinput.csv']`, is provided to Python.

3. Python will extract any *options* that are prior to the first *argument*. Options are distinguished from arguments by having a leading hyphen, `-`. These first options are used by Python during startup. In this example, there are no options. The first argument must be the Python filename that is to be run. This filename argument and all of the remaining words on the line will be assigned to `sys.argv`.
4. The Python startup is based on the options found. Depending on the `-s` option, the `site` module may be used to set up the import path, `sys.path`. If we used the `-m` option, then Python will use the `runpy` module to start our application. The given script files may be (re)compiled to byte code. The `-v` option will expose the imports that are being performed.
5. Our application can make use of `sys.argv` to parse options and arguments with the `argparse` module. Our application can use environment variables in `os.environ`. It can also parse configuration files; you can read Chapter 14, *Configuration Files and Persistence*, for more on this topic.

If there is no filename, the Python interpreter will read from standard input. If the standard input is a console (called a TTY, in Linux parlance), then Python will enter a **read-execute-print loop** (**REPL**) and display the `>>>` prompt. While we use this mode as developers, we don't generally make use of this mode for a finished application.

Because of Python's flexibility, there are some other ways of providing input to the Python runtime. The standard input can be a redirected file—for example, `python <some_file` or `some_app | python`. While both examples are valid uses of Python, they are potentially confusing because the application source is not very obvious.

Arguments and options

In order to run programs, the shell parses a command line into words. The words can be understood as a mixture of *options* and *arguments*. The following are some essential guidelines:

- Options come first. They are preceded by `-` or `--`. There are two formats: `-l` and `--word`. There are two species of options: options with no arguments and options with arguments. A couple of examples of options without arguments involve using `-v` to show a version or using `--version` to show the version. An example of an option with arguments is `-m module`, where the `-m` option must be followed by a module name.
- Short format (single-letter) options with no arguments can be grouped behind a single `-`. We might use `-bqv` to combine the `-b -q -v` options for convenience.
- Generally, arguments come after options, and they don't have a leading `-` or `--` (although some Linux applications break this rule). There are two common kinds of arguments:
    - Positional arguments, where the order is semantically significant. We might have two positional arguments: an input filename and an output filename. The order matters because the output file will be modified. When files will be overwritten, simply distinguishing by position needs to be done carefully to prevent confusion. The `cp`, `mv`, and `ln` commands are rare examples of positional arguments where the order matters. It's slightly more clear to use an option to specify the output file—for example, `-o output.csv`.
    - A list of arguments, all of which are semantically equivalent. We might have arguments that are all the names of input files. This fits nicely with the way the shell performs filename globing. When we say `process.py *.html`, the `*.html` command is expanded by the shell to filenames that become the positional parameters. (This doesn't work in Windows, so the `glob` module must be used.)

For more information, refer to: http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html#tag_12_02.

The Python command line has a dozen or so options that can control some details of Python's behavior. See the *Python Setup and Usage* document (`https://docs.python.org/3/using/index.html`) for more information on what these options are. The positional argument to the Python command is the name of the script that is to be run; this will be our application's topmost file.

Using the pathlib module

One of our principle forms of interaction with the OS is working with files. The `pathlib` module makes this particularly flexible. While the OS can represent a path to a file as a string, there is considerable syntactic subtlety to the strings that are used. Rather than try to parse the strings directly, it's much more pleasant to create `Path` objects. These can both compose and decompose paths from their constituent parts.

Path composition uses the `/` operator to assemble a `Path` from starting `Path` and `str` objects. This operator works for Windows as well as POSIX-compliant operating systems, such as Linux and macOS. Because a single operator will build appropriate paths, it's best to use `Path` objects for all filesystem access.

Here are some examples of building a `Path` object:

- `Path.home() / "some_file.dat"`: This names a given file in the user's home directory.
- `Path.cwd() / "data" / "simulation.csv"`: This names a file relative to the current working directory.
- `Path("/etc") / "profile"`: This names a file starting from the root of the filesystem.

There are a number of interesting inquiries that we can make to find details about a given `Path` object. In some cases, we might want to know a path's parent directory or the extension on a filename. Here are some examples:

```
p = Path.cwd() / "data" / "simulation.csv"
>>> p.parent
PosixPath('/Users/slott/mastering-oo-python-2e/data')
>>> p.name
'simulation.csv'
>>> p.suffix
'.csv'
>>> p.exists()
False
```

Note that these queries about a `Path` object do not depend on the path representing an actual object in the filesystem. In this example, the various properties of `parent`, `name`, and `suffix` are all reported correctly for a file that does not actually

exist. This is very useful for creating output filenames from input files.

For example, we might do the following:

```
>>> results = p.with_suffix('.json')
>>> results
PosixPath('/Users/slott/mastering-oo-python-2e/data/simulation.json')
```

We've taken an input `Path` object, `p`, and created an output `Path` object, `results`. The resulting object has the same name, but a different suffix. The new name was built by the `with_suffix()` method of a `Path`. This lets us create related files without having to parse the (relatively) complex path names.

As we noted in Chapter 6, *Using Callables and Contexts*, a file should be used as a context manager to ensure that it's closed properly. A `Path` object can open a file directly, leading to programs that work like the following example:

```
output = Path("directory") / "file.dat"
with output.open('w') as output_file:
    output_file.write("sample data\n")
```

This example creates a `Path` object. An OS will use a path without a leading `/` relative to the current working directory. The `open()` method of a `Path` will create a file object that can then be used for reading or writing. In this case, we're writing a constant string to a file.

We can use `Path` objects to manage directories as well as files. We often want to create working directories with code like the following:

```
>>> target = Path("data")/"ch18_directory"
>>> target.mkdir(exist_ok=True, parents=True)
```

We've assembled the `Path` object from a relative reference to the `data` directory and a specific subdirectory, `ch18_directory`. The `mkdir()` method of this `Path` object will ensure that the required directory structures are present in the filesystem. We've provided two common options. The `exists_ok` option will suppress the `FileExistsError` exception that would be raised if the file already exists. The `parents` option will create all of the required parent directories. This can be handy when creating complex, nested directory trees.

A common use case when working with web logs is to segregate the logs by date. We can create date-specific directories with code similar to the following

example:

```
>>> import datetime
>>> today = datetime.datetime.today()
>>> target = Path("data")/today.strftime("%Y%m%d")
>>> target.mkdir(exists_ok=True)
```

In this example, we've computed the current date. From this, we can create a directory path using the `data` subdirectory and the year, month, and day of the current date. We want to be tolerant of the directory that already exists, so we suppress the exception. This will not create parent directories, and if the data directory does not exist, there will be a `FileNotFoundError` exception raised.

Parsing the command line with argparse

The general approach to using `argparse` involves the following four steps:

1. First, we create an `ArgumentParser` instance. We can provide this object with overall information about the command-line interface. This might include a description, format changes for the displayed options and arguments, and whether or not `-h` is the `help` option. Generally, we only need to provide the description; the rest of the options have sensible defaults.
2. Then, we define the command-line options and arguments. This is done by adding arguments with the `ArgumentParser.add_argument()` method function.
3. Next, we parse the `sys.argv` command line to create a `namespace` object that details the options, option arguments, and overall command-line arguments.
4. Lastly, we use the `namespace` object to configure the application and process the arguments. There are a number of alternative approaches to handle this gracefully. This may involve parsing configuration files as well as command-line options. We'll look at several designs in this section.

An important feature of `argparse` is that it provides us with a unified view of options and arguments. The principle difference between the two is the number of times an option or argument value can occur. Options are—well, optional, and can occur one or no times. Arguments generally occur one or more times.

We can create a parser with code like the following:

```
parser = argparse.ArgumentParser(
    description="Simulate Blackjack")
```

We provided the description, as there's no good default value for that. Here are some common patterns to define the command-line API for an application:

- **A simple on–off option:** We'll often see this as a `-v` or `--verbose` option.
- **An option with an argument:** This might be a `-s ','` or `--separator '|'` option.
- **Any positional argument**: This might be used when we have an input file

and an output file as command-line arguments. This is rare, and should be avoided because it's never perfectly clear what the order should be.
- **All other arguments:** We'd use these when we have a list of input files.
- `--version`**:** This is a special option to display the version number and exit.
- `--help`**:** This option will display help and exit. This is a default, and so we don't need to do anything to make this happen.

Once the arguments have been defined, we can parse them and use them. Here's how we parse them:

```
config = parser.parse_args()
```

The `config` object is an `argparse.Namespace` object; the class is similar to `types.SimpleNamespace`. It will have a number of attributes, and we can easily add more attributes to this object.

We'll look at each of these six common kinds of arguments individually. There are a lot of clever and sophisticated parsing options available in the `ArgumentParser` class. Most of them go beyond the simplistic guidelines commonly suggested for command-line argument processing. In general, we should avoid the kind of super-complex options that characterize programs such as `find`. When command-line options get terribly complex, we may have drifted into creating a domain-specific language on top of Python. This often means we're creating a kind of framework, not simply creating an application.

A simple on–off option

We will define a simple on–off option with a one-letter short name or a longer name. We should also provide an explicit action. We might want to provide a destination variable if we omit the longer name, or if the longer name is unpleasant as a Python variable. Let's set this up using the following code:

```
parser.add_argument(
    '-v', '--verbose', action='store_true', default=False)
```

This will define the long and short versions of the command-line option. If the option is present, the action will set the `verbose` option to `True`. If the option is absent, the `verbose` option will default to `False`. Here are some common variations of this theme:

- We might change the action to `'store_false'` with a default of `True`.
- Sometimes, we'll have a default of `None` instead of `True` or `False`.
- Sometimes, we'll use an action of `'store_const'` with an additional `const=` argument. This allows us to move beyond simple Boolean values and store things such as logging levels or other objects.
- We might also have an action of `'count'`, which allows the option to get repeated, increasing the count. In this case, the default is often zero.

If we're using the logger, we might define a debugging option like the following code:

```
parser.add_argument(
    '--debug', action='store_const', const=logging.DEBUG,
    default=logging.INFO, dest="logging_level" )
```

We changed the action to `store_const`, which stores a constant value and provides a specific constant value of `logging.DEBUG`. This means that the resulting options object will directly provide the value needed to configure the root logger. We can then simply configure the logger using `config.logging_level` without any further mapping or conditional processing.

An option with an argument

We'll define an option that has an argument with the long and optional short name. We'll provide an action that stores the value provided with the argument. We can also provide a type conversion, in case we want `float` or `int` values instead of a string. Let's set this up using the following code:

```
parser.add_argument(
    "-b", "--bet", action="store", default="Flat",
    choices=["Flat", "Martingale", "OneThreeTwoSix"],
    dest='betting_rule')
parser.add_argument(
    "-s", "--stake", action="store", default=50, type=int)
```

The first example will define two versions of the command-line syntax, both long and short version. When parsing the command-line argument values, a string value must follow the option, and it must be from the available choices. The destination name, `betting_rule`, will receive the option's argument string.

The second example also defines two versions of the command-line syntax. It includes a type conversion. When parsing argument values, this will store an integer value that follows the option. The long name, `stake`, will be the value in the options object created by the parser.

In some cases, there may be a list of values associated with the argument. In this case, we may provide a `nargs="+"` option to collect multiple values separated by spaces in a list.

Positional arguments

We define positional arguments using a name with no "`-`" decoration. In a case where we have a fixed number of positional arguments, we'll add them appropriately to the parser, as shown in the following code:

```
parser.add_argument("input_filename", action="store")
parser.add_argument("output_filename", action="store")
```

When parsing argument values, the two positional argument strings will be stored in the final namespace object. We can use `config.input_filename` and `config.output_filename` to work with these argument values.

As noted previously, using simple positional parameters to identify output files can cause problems for users. The GNU/Linux `cp`, `mv`, and `ln` programs should be considered *exceptional* cases where a file can get overwritten. The preferred approach is to use an option with a value to specify the files that can be destroyed. It's almost always safer to force users to use an option like `-o name.csv` to make it *perfectly* clear what the output filename will be.

TIP

Avoid defining commands where the exact order of arguments is significant.

All other arguments

We define argument lists with a name that has no `-` decoration and a piece of advice in the `nargs=` parameter. If the rule is one or more argument values, we specify `nargs="+"`. If the rule is zero or more argument values, we specify `nargs="*"`, as shown in the following code. If the rule is *optional*, we specify `nargs="?"`. This will collect all other argument values into a single sequence in the resulting namespace:

```
parser.add_argument(
    "filenames", action="store", nargs="*", metavar="file...")
```

When the list of filenames is optional, it generally means that `STDIN` or `STDOUT` will be used if no specific filenames are provided.

If we specify a `nargs=` value, then the result becomes a list. If we specify `nargs=1`, then the resulting object is a one-element list; this generalizes nicely if we need to change to `nargs='+'`. As a special case, omitting `nargs`, leads to a result that is a single value; this does not generalize well.

Creating a list (even if it has only one element) is handy because we might want to process the arguments in the following manner:

```
for filename in config.filenames:
    process(filename)
```

In some cases, we may want to provide a sequence of input files that includes `STDIN`. The common convention for this is a filename of `-` as an argument. We'll have to handle this within our application with something like the following code:

```
for filename in config.filenames:
    if filename == '-':
        process(sys.stdin)
    else:
        with open(filename) as input:
            process(input)
```

This shows us a loop that will attempt to handle a number of filenames, potentially including `-` to show when to process standard input among a list of

files. A `try:` block should probably be used around the `with` statement.

--version display and exit

The option to display the version number is so common that there's a special shortcut to show us the version information:

```
parser.add_argument(
    "-V", "--version", action="version", version=__version__ )
```

This example assumes that we have a global `__version__= "3.3.2"` module somewhere in the file. This special `action="version"` will have the side effect of exiting the program after displaying the version information.

--help display and exit

An option to display help is a default feature of `argparse`. Another special case allows us to change the `help` option from the defaults to`-h` or `--help`. This requires two things. First, we must create the parser with `add_help=False`. This will turn off the built-in `-h`, `--help` feature. After doing that, we will add the argument that we want to use (for example, `'-?'`) with `action="help"`. This will display the help text and exit.

Integrating command-line options and environment variables

The general policy for environment variables is to provide configuration inputs, similar to command-line options and arguments. For the most part, we use environment variables for settings that rarely change. We'll often set them via the `.bashrc` or `.bash_profile` files so that the values are set every time we log in. We may set the environment variables more globally in an `/etc/bashrc` file so that they apply to all users. We can also set environment variables on the command line, but these settings only apply to the program being run.

In some cases, all of our configuration settings can be provided on the command line. In this case, the environment variables could be used as a kind of backup syntax for slowly changing variables.

In other cases, the configuration values providing environment variables may be disconnected from the configuration performed by command-line options. We may need to get some values from the environment and merge in values that come from the command line.

We can leverage environment variables to set the default values in a configuration object. We want to gather these values prior to parsing the command-line arguments. This way, command-line arguments can override environment variables. There are two common approaches to this:

- **Explicitly setting the values when defining the command-line options**: This has the advantage of making the default value show up in the help message. It only works for environment variables that overlap with command-line options. We might do something like the following to use the `SIM_SAMPLES` environment variable to provide a default value that can be overridden:

```
parser.add_argument(
    "--samples",
    action="store",
    default=int(os.environ.get("SIM_SAMPLES",100)),
    type=int,
```

```
    help="Samples to generate")
```

- **Implicitly setting the values as part of the parsing process**: This makes it simple to merge environment variables with command-line options into a single configuration. We can populate a namespace with default values and then overwrite it with the parsed values from the command line. This provides us with three levels of option values: the default defined in the parser, an override value seeded into the namespace, and finally, any override value provided on the command line, as shown in the following code:

```
config4 = argparse.Namespace()
config4.samples = int(os.environ.get("SIM_SAMPLES",100))
config4a = parser.parse_args(sys.argv[1:], namespace=config4)
```

The argument parser can perform type conversions for values that are not simple strings. However, gathering environment variables doesn't automatically involve a type conversion. For options that have non-string values, we must perform the type conversion in our application.

Providing more configurable defaults

We can incorporate configuration files along with environment variables and the command-line options. This gives us three ways to provide a configuration to an application program:

- A hierarchy of configuration files can provide default values. See `Chapter 14`, *Configuration Files and Persistence*, for examples of the various ways to do this.
- Environment variables can provide overrides to the configuration files. This may mean translating from an environment variable namespace to the configuration namespace.
- The command-line options define the final overrides.

Using all three may be too much of a good thing. Tracking down a setting can become difficult if there are too many places to search. The final decision about the configuration often rests on staying consistent with the overall collection of applications and frameworks. We should strive to make our programming fit seamlessly with other components.

Overriding configuration file settings with environment variables

We'll use a three-stage process to incorporate environment variables. For this application, the environment variables will be used to override configuration file settings. The first stage is to gather the default values from the various files. This is based on the examples shown in `Chapter 14`, *Configuration Files and Persistence*. We can use code like the following:

```
config_locations = (
    Path.cwd(),
    Path.home(),
    Path.cwd() / "opt",  # A testing stand-in for Path("/opt")
    Path(__file__) / "config",
    # Other common places...
    # Path("~someapp").expanduser(),
)

candidate_paths = (dir / "ch18app.yaml" for dir in config_locations)
config_paths = (path for path in candidate_paths if path.exists())
files_values = [yaml.load(str(path)) for path in config_paths]
```

This example uses a sequence of locations, ranked in order of importance. The value in the current working directory provides the most immediate configuration. For values not set here, the user's home directory is a place to keep general settings. We should use an `opt` subdirectory of the current working directory, `Path.cwd()/"opt"`; this stands in place of `Path("/etc")` or `Path("/opt")`. A standard name, `"ch18app.yaml"`, is put after the various directory paths to create a number of concrete paths for configuration files to set the `candidate_paths` variable. A generator expression assigned to `config_paths` will yield an iterable sequence of paths that actually exists.

The final result in `files_values` is a sequence of configuration values taken from the files that are found to exist. Each file should create a dictionary that maps parameter names to parameter values. This list can be used as part of a final `ChainMap` object.

The second stage is to build the user's environment-based settings. We can use code like the following to set this up:

```
env_settings = [
    ("samples", nint(os.environ.get("SIM_SAMPLES", None))),
    ("stake", nint(os.environ.get("SIM_STAKE", None))),
    ("rounds", nint(os.environ.get("SIM_ROUNDS", None))),
]
env_values = {k: v for k, v in env_settings if v is not None}
```

Creating a mapping like this has the effect of rewriting external environment variable names like `SIM_SAMPLES` into internal configuration names like `samples`. Internal names will match our application's configuration attributes. External names are often defined in a way that makes them unique in a complex environment.

For environment variables that were not defined, the `nint()` function, shown in the following code, will provide `None` as a default value if the environment variable is not defined. When we create the `env_values`, the `None` objects are removed from the initial collection of environment values.

Given a number of dictionaries, we can use `ChainMap` to combine them, as shown in the following code:

```
defaults = argparse.Namespace(
    **ChainMap(
        env_values,  # Checks here first
        *files_values  # All of the files, in order
    )
)
```

We combined the various mappings into a single `ChainMap`. The environment variables are searched first. When values are present there, the values are looked up from the user's configuration file first and then other configurations, if the user configuration file didn't provide a value.

The `*files_values` ensures that the list of values will be provided as a sequence of positional argument values. This allows a single sequence (or iterable) to provide values for a number of positional parameters. `**ChainMap` ensures that a dictionary is turned into a number of named parameter values. Each key becomes a parameter name associated with the value from the dictionary. Here's an example of how this works:

```
>>> argparse.Namespace(a=1, b=2)
Namespace(a=1, b=2)
>>> argparse.Namespace(**{'a': 1, 'b': 2})
Namespace(a=1, b=2)
```

The resulting `Namespace` object can be used to provide defaults when parsing the command-line arguments. We can use the following code to parse the command-line arguments and update these defaults:

```
config = parser.parse_args(sys.argv[1:], namespace=defaults)
```

We transformed our `ChainMap` of configuration file settings into an `argparse.Namespace` object. Then we parsed the command-line options to update that namespace object. As the environment variables are first in `ChainMap`, they override any configuration files.

Making the configuration aware of the None values

This three-stage process to set the environment variables includes many common sources of parameters and configuration settings. We don't always need environment variables, configuration files, and command-line options; some applications may only need a subset of these techniques.

We often need type conversions that will preserve `None` values. Keeping the `None` values will ensure that we can tell when an environment variable was not set. Here's a more sophisticated type conversion that can be called None-aware:

```
from typing import Optional

def nint(x: Optional[str]) -> Optional[int]:
    if x is None:
        return x
    return int(x)
```

We use this when converting environment variable values to integers. If an environment variable is not set, a default of `None` will be used. If the environment variable is set, then the value will be converted to an integer. In later processing steps, we can depend on the `None` value to build a dictionary from only the proper values that are not `None`.

We can use similar None-aware conversions to handle `float` values. We don't need to do any conversion for strings, and `os.environ.get("SIM_NAME")` will provide the environment variable value or `None`.

Customizing the help output

Here's some typical output that comes directly from the default `argparse.print_help()` code:

```
usage: ch18_ex1.py [-v] [--debug] [--dealerhit {Hit17,Stand17}]
                   [--resplit {ReSplit,NoReSplit,NoReSplitAces}]
                   [--decks DECKS] [--limit LIMIT] [--payout PAYOUT]
                   [-p {SomeStrategy,AnotherStrategy}]
                   [-b {Flat,Martingale,OneThreeTwoSix}] [-r ROUNDS]
                   [-s STAKE] [--samples SAMPLES] [-V] [-?]
                   output

Simulate Blackjack

positional arguments:
  output

optional arguments:
  -v, --verbose
  --debug
  --dealerhit {Hit17,Stand17}
  --resplit {ReSplit,NoReSplit,NoReSplitAces}
  --decks DECKS Decks to deal (default: 6)
  --limit LIMIT
  --payout PAYOUT
  -p {SomeStrategy,AnotherStrategy}, --playerstrategy {SomeStrategy,AnotherStrategy}
  -b {Flat,Martingale,OneThreeTwoSix}, --bet {Flat,Martingale,OneThreeTwoSix}
  -r ROUNDS, --rounds ROUNDS
  -s STAKE, --stake STAKE
  --samples SAMPLES Samples to generate (default: 100)
  -V, --version show program's version number and exit
  -?, --help
```

The default help text is built from four things in our parser definition:

- The `usage:` line is a summary of the options. We can replace the default calculation with our own usage text that omits the less commonly used details.
- This is followed by the description. By default, the text we provide is cleaned up a bit. In this example, we provided a shabby two-word description, `Simulate Blackjack`, so there's no obvious cleanup.
- Then, the arguments are shown. These come in two subgroups:
  - The positional arguments
  - The options, in the order that we defined them.
- After this, an optional epilogue text may be shown; we didn't provide any for this definition.

In some cases, this kind of terse reminder is adequate. In other cases, however, we may need to provide more details. We have three tiers of support for more detailed help:

- **Add `help=` to the argument definitions**: This is the place to start when customizing the help details. This will supplement the option description with more meaningful details.
- **Use one of the other help formatter classes**: This is done with the `formatter_class=` argument when building `ArgumentParser`. If we want to use `ArgumentDefaultsHelpFormatter`, then this works with the `help=` values for each argument definition.
- **Extend the `ArgumentParser` class and override the `print_usage()` and `print_help()` methods**: This allows us to write very sophisticated output. This should not be used casually. If we have options so complex that ordinary help features don't work, then perhaps we've gone too far.

Our goal is to improve usability. Even if our programs work correctly, we can build trust by providing command-line support that makes our program easier to use.

Creating a top-level main() function

In Chapter 14, *Configuration Files and Persistence*, we suggested two application configuration design patterns:

- **A global property map**: In the previous examples, we implemented the global property map with a `Namespace` object created by `ArgumentParser`.
- **Object construction**: The idea behind object construction was to build the required object instances from the configuration parameters, effectively demoting the global property map to a local property map inside the `main()` function and not saving the properties.

What we showed you in the previous section was the use of a local `Namespace` object to collect all of the parameters. From this, we can build the necessary application objects that will do the real work of the application. The two design patterns aren't a dichotomy; they're complementary. We used `Namespace` to accumulate a consistent set of values and then built the various objects based on the values in that namespace.

This leads us to a design for a top-level function. Before looking at the implementation, we need to consider a proper name for this function. There are two ways to name the function:

- Name it `main()`, because that's a common term for the starting point of the application as a whole; everyone expects this.
- Don't name it `main()`, because `main()` is too vague to be meaningful in the long run and limits reuse. If we follow this path, we can make a meaningful top-level function with a name that's a `verb_noun()` phrase to describe the operation fairly. We can also add a line `main = verb_noun` that provides an alias of `main()`.

Using the second, two-part implementation, lets us change the definition of `main()` through extension. We can add a new function and reassign the name `main` to the newer function. Old function names are left in place as part of a stable, growing API.

Here's a top-level application script that builds objects from a configuration `Namespace` object:

```
import ast
import csv
import argparse

def simulate_blackjack(config: argparse.Namespace) -> None:
    dealer_classes = {"Hit17": Hit17, "Stand17": Stand17}
    dealer_rule = dealer_classes[config.dealer_rule]()
    split_classes = {
        "ReSplit": ReSplit, "NoReSplit": NoReSplit, "NoReSplitAces": NoReSplitAces
    }
    split_rule = split_classes[config.split_rule]()
    try:
        payout = ast.literal_eval(config.payout)
        assert len(payout) == 2
    except Exception as ex:
        raise ValueError(f"Invalid payout {config.payout}") from ex
    table = Table(
        decks=config.decks,
        limit=config.limit,
        dealer=dealer_rule,
        split=split_rule,
        payout=payout,
    )
    player_classes = {"SomeStrategy": SomeStrategy, "AnotherStrategy": AnotherStrategy}
    player_rule = player_classes[config.player_rule]()
    betting_classes = {
        "Flat": Flat, "Martingale": Martingale, "OneThreeTwoSix": OneThreeTwoSix
    }
    betting_rule = betting_classes[config.betting_rule]()
    player = Player(
        play=player_rule,
        betting=betting_rule,
        max_rounds=config.rounds,
        init_stake=config.stake,
    )
    simulate = Simulate(table, player, config.samples)
    with Path(config.outputfile).open("w", newline="") as target:
        wtr = csv.writer(target)
        wtr.writerows(simulate)

main = simulate_blackjack
```

The `simulate_blackjack` function depends on an externally supplied `Namespace` object with the configuration attributes. It's not named `main()` so that we can make future additions and changes. We can reassign `main` to any new function that replaces or extends this function.

This function builds the various objects—`Table`, `Player`, and `Simulate`—that are required. We configured these objects based on the supplied configuration parameters.

We've set up the object that does the real work. After the object construction, the

actual work is a single, highlighted line: `wtr.writerows(simulate)`. About 90 percent of the program's time will be spent here, generating samples and writing them to the required file.

A similar pattern holds for GUI applications. They enter a main loop to process GUI events. The pattern also holds for servers that enter a main loop to process requests.

We've depended on having a configuration object passed in as an argument. This follows from our testing strategy of minimizing dependencies. This top-level `simulate_blackjack()` function doesn't depend on the details of how the configuration was created. We can then use this function in an application script, as follows:

```
if __name__ == "__main__":
    logging.config.dictConfig(yaml.load("logging.config"))
    config5 = get_options_2(sys.argv[1:])
    simulate_blackjack(config5)
    logging.shutdown()
```

This represents a separation of concerns. The work of the application is separated into three separate parts:

- The outermost level is defined by logging. We configured logging outside of all other application components to ensure that there are no conflicts between other top-level packages configuring logging. When we look at combining applications into larger composite processing, we need to be sure that several applications being combined doesn't result in conflicting logging configurations.
- The inner level is defined by the application's configuration. We don't want conflicts among separate application components. We'd like to allow a single command-line API to evolve separately from our application implementations. We'd like to be able to embed our application processing into separate environments, perhaps defined by `multiprocessing` or a RESTful web server.
- The final portion is the `simulate_blackjack()` function. This is separated from the logging and configuration issues. This allows a variety of techniques to be used to provide a configuration of parameters. Furthermore, when we look at combining this with other processing, the separation of logging and configuration will be helpful.

Ensuring DRY for the configuration

We have a potential **Don't Repeat Yourself** (**DRY**) issue between our construction of the argument parser and the use of the arguments to configure the application. We built the arguments using some keys that are repeated.

We can eliminate this repetition by creating some internal configurations that map to the externally visible values. For example, we might define this global as follows:

```
dealer_rule_map = {"Hit17": Hit17, "Stand17", Stand17}
```

We can use it to create the argument parser, as follows:

```
parser.add_argument(
    "--dealerhit", action="store", default="Hit17",
    choices=dealer_rule_map.keys(),
    dest='dealer_rule')
```

We can use it to create the working objects, as follows:

```
dealer_rule = dealer_rule_map[config.dealer_rule]()
```

This eliminates the repetition. It allows us to add new class definitions and parameter key mappings in one place as the application evolves. It also allows us to abbreviate or otherwise rewrite the external API, as shown here:

```
dealer_rule_map ={"H17": Hit17, "S17": Stand17}
```

There are four of these kinds of mappings from the command-line (or configuration file) string to the application class. Using these internal mappings simplifies the `simulate_blackjack()` function.

Managing nested configuration contexts

In a way, the presence of nested contexts means that top-level scripts ought to look like the following code:

```
if __name__ == "__main__":
    with Setup_Logging():
        with Build_Config(arguments) as config_3:
            simulate_blackjack_betting(config_3)
```

We've added two context managers to formalize the creation of the working context. For more information, see Chapter 6, *Using Callables and Contexts*. Here is a context manager for logging:

```
class Setup_Logging:
    def __enter__(self, filename="logging.config") -> "Setup_Logging":
       logging.config.dictConfig(yaml.load(filename))

    def __exit__(self, *exc) -> None:
       logging.shutdown()
```

The idea is to be sure that the logging process is properly configured and shut down when the application runs. This removes any doubts about whether or not a final buffer was saved and the log file was properly closed.

Similarly, we can define a context manager to build the configuration required to run the application. In this case, the context manager is a very thin wrapper around the `get_options_2()` function, as shown in the preceding code. The context manager looks like this:

```
from typing import List

class Build_Config:

    def __init__(self, argv: List[str]) -> None:
        self.options = get_options_2(argv)

    def __enter__(self) -> argparse.Namespace:
        return self.options

    def __exit__(self, *exc) -> None:
        return
```

The `Build_Config` context manager can gather the configuration from a number of files, as well as command-line arguments. The use of a context manager isn't essential in this case; however, it leaves room for extension in case the configuration becomes more complex.

This design pattern may clarify the various concerns that surround the application startup and shutdown. While it may be a bit much for most applications, the essential fit with the Python context managers seems like it could be helpful as an application grows and expands.

When we're confronted with an application that grows and expands, we often wind up doing larger-scale programming. For this, it's important to separate the changeable application processing from the less changeable processing context.

Programming in the large

Let's add a feature to our blackjack simulation: analysis of results. We have several paths to implement this added feature. There are two dimensions to the considerations that we must make, leading to a large number of combinations. One dimension is how to design the new features:

- We can add a function and work out ways to integrate it into the whole.
- We can use the Command design pattern and create a hierarchy of commands, some of which are single functions and others of which are sequences of functions.

Another dimension of the design that we must consider is how to package the new features:

- We can write a new top-level script file. This tends to create new commands based on filenames. We might start with commands such as `simulate.py` and `analyze.py`.
- We can add a parameter to an application that allows one script to perform either the simulation or the analysis, or both. We would have commands that look like `app.py simulate` and `app.py analyze`.

This leads to four combinations of implementation choices. We'll focus on using the Command design pattern. First, we'll revise our existing application to use the Command design pattern. Then, we'll extend our application by adding features in the form of new command subclasses.

Designing command classes

Many applications involve an implicit Command design pattern; after all, we're *processing* data. To do this, there must be at least one active-voice verb, or command, that defines how the application transforms, creates, or consumes data. A simple application may have only a single verb, implemented as a function. Using the command class design pattern may not be helpful for simple applications.

More complex applications will have multiple, related verbs. One of the key features of GUIs and web servers is that they can do multiple things, leading to multiple commands. In many cases, the GUI menu options define the domain of the verbs for an application.

In some cases, an application's design stems from a decomposition of a larger, more complex verb. We may factor the overall processing into several smaller command steps that are combined in the final application.

When we look at the evolution of an application, we often see a pattern where new functionality is accreted. In these cases, each new feature can become a kind of separate command subclass that is added to the application class hierarchy.

An abstract superclass for commands has the following design:

```
class Command:

    def __init__(self) -> None:
        self.config: Dict[str, Any] = {}

    def configure(self, namespace: argparse.Namespace) -> None:
        self.config.update(vars(namespace))

    def run(self) -> None:
        """Overridden by a subclass"""
        pass
```

We configure this `Command` class by setting the `config` property to `argparse.Namespace`. This will populate the instance variables from the given `namespace` object.

Once the object is configured, we can set it to doing the work of the command

by calling the `run()` method. This class implements a relatively simple use case, as shown in the following code:

```
    main = SomeCommand()
    main.configure = some_config
    main.run()
```

This captures the general flavor of creating an object, configuring it, and then letting it do the work it was configured for. We can expand on this idea by adding features to the command subclass definition.

Here's a concrete subclass that implements a blackjack simulation:

```
class Simulate_Command(Command):
    dealer_rule_map = {
        "Hit17": Hit17, "Stand17": Stand17}
    split_rule_map = {
        "ReSplit": ReSplit, "NoReSplit": NoReSplit, "NoReSplitAces": NoReSplitAces
    }
    player_rule_map = {
        "SomeStrategy": SomeStrategy, "AnotherStrategy": AnotherStrategy}
    betting_rule_map = {
        "Flat": Flat, "Martingale": Martingale, "OneThreeTwoSix": OneThreeTwoSix
    }

    def run(self) -> None:
        dealer_rule = self.dealer_rule_map[self.config["dealer_rule"]]()
        split_rule = self.split_rule_map[self.config["split_rule"]]()
        payout: Tuple[int, int]
        try:
            payout = ast.literal_eval(self.config["payout"])
            assert len(payout) == 2
        except Exception as e:
            raise Exception(f"Invalid payout {self.config['payout']!r}") from e
        table = Table(
            decks=self.config["decks"],
            limit=self.config["limit"],
            dealer=dealer_rule,
            split=split_rule,
            payout=payout,
        )
        player_rule = self.player_rule_map[self.config["player_rule"]]()
        betting_rule = self.betting_rule_map[self.config["betting_rule"]]()
        player = Player(
            play=player_rule,
            betting=betting_rule,
            max_rounds=self.config["rounds"],
            init_stake=self.config["stake"],
        )
        simulate = Simulate(table, player, self.config["samples"])
        with Path(self.config["outputfile"]).open("w", newline="") as target:
            wtr = csv.writer(target)
            wtr.writerows(simulate)
```

This class implements the essential top-level function that configures the various objects and then executes the simulation. This class refactors the

`simulate_blackjack()` function shown previously to create a concrete extension of the `Command` class. This can be used in the main script, as shown in the following code:

```
if __name__ == "__main__":
    with Setup_Logging():
        with Build_Config(sys.argv[1:]) as config:
            main = Simulate_Command()
            main.configure(config)
            main.run()
```

While we could make this command into `Callable` and use `main()` instead of `main.run()`, the use of a callable can be confusing. We're explicitly separating the following three design issues:

- **Construction**: We've specifically kept the initialization empty. In a later section, we'll show you some examples of PITL, where we'll build a larger composite command from smaller component commands.
- **Configuration**: We've put the configuration in via a `property` setter, isolated from the construction and control.
- **Control**: This is the real work of the command after it's been built and configured.

When we look at a callable or a function, the construction is part of the definition. The configuration and control are combined into the function call itself. We sacrifice a small bit of flexibility if we try to define a callable.

Adding the analysis command subclass

We'll extend our application by adding the analysis feature. As we're using the Command design pattern, we can add yet another subclass for analysis.

Here's our analysis feature, also designed as a subclass of the `Command` class:

```
class Analyze_Command(Command):

    def run(self) -> None:
        with Path(self.config["outputfile"]).open() as target:
            rdr = csv.reader(target)
            outcomes = (float(row[10]) for row in rdr)
            first = next(outcomes)
            sum_0, sum_1 = 1, first
            value_min = value_max = first
            for value in outcomes:
                sum_0 += 1  # value**0
                sum_1 += value  # value**1
                value_min = min(value_min, value)
                value_max = max(value_max, value)
            mean = sum_1 / sum_0
            print(
                f"{self.config['outputfile']}\n"
                f"Mean = {mean:.1f}\n"
                f"House Edge = {1 - mean / 50:.1%}\n"
                f"Range = {value_min:.1f} {value_max:.1f}"
            )
```

This class inherits the general features of the `Command` class. In this case, there's only one small feature, which is to save the configuration information. The work performed by the `run()` method is not too statistically meaningful—true, but the point is to show you a second command that uses the configuration namespace to do work related to our simulation. We used the `outputfile` configuration parameter to name the file that is read to perform some statistical analysis.

Adding and packaging more features into an application

Previously, we noted one common approach to supporting multiple features. Some applications use multiple top-level main programs in separate `.py` script files. If we do this, then combining commands from separate files forces us to write a shell script. It doesn't seem optimal to introduce yet another tool and another language to do **programming in-the-large** (**PITL**).

A slightly more flexible alternative to creating separate script files is using a positional parameter to select a specific top-level `Command` object. For our example, we'd like to select either the simulation or the analysis command. To do this, we would add a parameter to the command-line argument parsing the following code:

```
parser.add_argument(
    "command", action="store", default='simulate',
    choices=['simulate', 'analyze'])
parser.add_argument("outputfile", action="store", metavar="output")
```

This would change the command-line API to add the top-level verb to the command line. We can then map our argument values to class names that implement the desired command, as shown in the following code:

```
command_map = {
    'simulate': Simulate_Command,
    'analyze': Analyze_Command
}
command = command_map[options.command]
command.configure(options)
command.run()
```

This allows us to create even higher-level composite features. For example, we might want to combine simulation and analysis into a single, overall program. We also might like to do this without resorting to using the shell.

Designing a higher-level, composite command

We can also design a composite command that's built from other commands. For this, we have two design strategies: object composition and class composition.

If we use object composition, then our composite command is based on the built-in `list` or `tuple`. We can extend or wrap one of the existing sequences. We'll create the composite `Command` object as a collection of instances of other `Command` objects. We might consider writing something like the following code:

```
simulate_and_analyze = [Simulate(), Analyze()]
```

This has the disadvantage that we haven't created a new class for our unique composite command. We created a generic composite and populated it with instances. If we want to create even higher-level compositions, we'll have to address this asymmetry between low-level `Command` classes and higher-level composite `Command` objects based on built-in sequence classes.

We'd prefer to have a composite command that was also a subclass of a command. If we use class composition, then we'll have a more consistent structure for our low-level commands and our higher-level composite commands.

Here's a class that implements a sequence of other commands:

```
class Command_Sequence(Command):
    steps: List[Type[Command]] = []

    def __init__(self) -> None:
        self._sequence = [class_() for class_ in self.steps]

    def configure(self, config: argparse.Namespace) -> None:
        for step in self._sequence:
            step.configure(config)

    def run(self) -> None:
        for step in self._sequence:
            step.run()
```

We defined a class-level variable, `steps`, to contain a sequence of command

classes. During the object initialization, `__init__()` will construct an internal instance variable, `_sequence`, with objects of the named classes in `self.steps`.

When the configuration is set, it will be pushed into each constituent object. When the composite command is executed via `run()`, it is delegated to each component in the composite command.

Here's a `Command` subclass built from two other `Command` subclasses:

```
class Simulate_and_Analyze(Command_Sequence):
    steps = [Simulate_Command, Analyze_Command]
```

This class is only a single line of code to define the sequence of steps. As this is a subclass of the `Command` class itself, it has the necessary polymorphic API. We can now create compositions with this class because it's compatible with all other subclasses of `Command`.

We can now make the following very small modification to the argument parsing to add this feature to the application:

```
parser.add_argument(
    "command", action="store", default='simulate',
    choices=['simulate', 'analyze', 'simulate_analyze']
)
```

We simply added another choice to the argument option values. We'll also need to tweak the mapping from the argument option string to the class, as follows:

```
command_map = {
    'simulate': Simulate_Command,
    'analyze': Analyze_Command,
    'simulate_analyze': Simulate_and_Analyze}
```

Note that we shouldn't use a vague name such as `both` to combine two commands. If we avoid vagueness, we create opportunities to expand or revise our application. Using the Command design pattern makes it pleasant to add features. We can define composite commands, or we can decompose a larger command into smaller subcommands.

Packaging and implementation may involve adding an option choice and mapping that choice to a class name. If we use a more sophisticated configuration file (see Chapter 14, *Configuration Files and Persistence*), we can provide the class name directly in the configuration file and save the mapping

from an option string to a class.

Additional composite Command design patterns

We can identify a number of composite Command design patterns. In the previous example, we designed a composite object that implemented a sequence of operations. For inspiration, we can look at the bash shell composite operators: ;, &, |, as well as () for grouping. Beyond these, we have if, for, and while loops within the shell.

We looked at the semantic equivalent of the shell sequence operator, ;, in the Command_Sequence class definition. This concept of a *sequence* is so ubiquitous that many programming languages (such as the shell and Python) don't require an explicit operator; the shell's syntax simply uses end-of-line as an implied sequence operator.

The shell's & operator creates two commands that run concurrently instead of sequentially. We can create a Command_Concurrent class definition with a run() method that uses multiprocessing to create two subprocesses and waits for both to finish.

The | operator in the shell creates a pipeline: one command's output buffer is another command's input buffer—the commands run concurrently. In Python, we'd need to create a queue as well as two processes to read and write that queue. This is a more complex situation: it involves populating the queue objects into the configurations of each of the various children. Chapter 13, *Transmitting and Sharing Objects*, has some examples of using multiprocessing with queues to pass objects among concurrent processes.

The if command in the shell has a number of use cases; however, there's no compelling reason to provide anything more than a native Python implementation via a method in a subclass of Command. Creating a complex Command class to mimic Python's if-elif-else processing isn't helpful; we can—and should—use Python directly.

The while and for commands in the shell, similarly, aren't the sorts of things we

need to define in a higher-level `Command` subclass. We can simply write this in a method in Python.

Here's an example of a *for-all* class definition that applies an existing command to all the values in a collection:

```
class ForAllBets_Simulate(Command):

    def run(self) -> None:
        for bet_class in "Flat", "Martingale", "OneThreeTwoSix":
            self.config["betting_rule"] = bet_class
            self.config["outputfile"] = Path("data")/f"ch18_simulation7_{bet_class}.dat"
            sim = Simulate_Command()
            sim.configure(argparse.Namespace(**self.config))
            sim.run()
```

We enumerated the three classes of betting in our simulation. For each of these classes, we tweaked the configuration, created a simulation, and executed that simulation.

Note that this *for-all* class won't work with the `Analyze_Command` class defined previously. We can't simply create composites that reflect different scopes of work. The `Analyze_Command` class runs a single simulation, but the `ForAllBets_Simulate` class runs a collection of simulations. We have two choices to create compatible scopes of work: we could create an `Analyze_All` command or a `ForAllBets_Sim_and_Analyze` command. The design decision depends on the needs of the users.

Integrating with other applications

There are several ways in which we can use Python when integrating with other applications. It's difficult to provide a comprehensive overview, as there are so many applications, each with unique, distinctive features. We can show you some broad design patterns in the following list:

- Python can be the application's scripting language. You can find a list of applications that simply include Python as the primary method to add features at `https://wiki.python.org/moin/AppsWithPythonScripting`.
- A Python module can implement the application's API. There are numerous applications that include Python modules that provide a binding to the application's API. Application developers working in one language will often provide API libraries for other languages, including Python.
- We can use the `ctypes` module to implement another application's API directly in Python. This works out well in the case of an application library that is focused on C or C++.
- We can use `sys.stdin` and `sys.stdout` to create a shell-level pipeline that connects us to another application. We might also want to look at the `fileinput` module when building shell-compatible applications.
- We can use the `subprocess` module to access an application's command-line interface. This may also involve connecting to an application's `stdin` and `stdout` to interact properly with it.
- We can also write our own Python-compatible module in C or C++. In this case, we can implement the foreign application's API in C, offering classes or functions that a Python application can leverage. This may offer better performance than using the `ctypes` API. As this requires compiling C or C++, it's also a bit more tool intensive.

This level of flexibility means that we often use Python as the integration framework or glue to create a larger, composite application from smaller applications. When using Python for integration, we'll often have Python classes and objects that mirror the definitions in another application.

There are some additional design considerations that we'll save for `Chapter 19`, *Modules and Package Design*. These are higher-level architectural design

considerations, above and beyond coping with the command line.

Summary

In this chapter, we looked at how to use `argparse` and `os.environ` to gather command-line argument and configuration parameters. This builds on the techniques shown in `Chapter 14`, *Configuration Files and Persistence*.

We learned how to implement a number of common command-line features using `argparse`. This includes common features, such as showing the version number and exiting or showing the help text and exiting.

We looked at using the Command design pattern to create applications that can be expanded or refactored to offer new features. Our goal is to explicitly keep the body of the top-level main function as small as possible.

Design considerations and trade-offs

The command-line API is an important part of a finished application. While most of our design effort focuses on what the program does while it's running, we do need to address two boundary states: startup and shutdown. An application must be easy to configure when we start it up. It must also shut down gracefully, properly flushing all of the output buffers and releasing all of the OS resources.

When working with a public-facing API, we have to address a variation on the problem of schema evolution. As our application evolves—and as our knowledge of the users evolves—we will modify the command-line API. This may mean that we'll have legacy features or legacy syntax. It may also mean that we need to break the compatibility with the legacy command-line design.

In many cases, we'll need to be sure that the major version number is part of our application's name. We shouldn't write a top-level module named `someapp`. When we need to make major release three, which is incompatible with major release two, we may find it awkward to explain that the name of the application has changed to `someapp3`. We should consider starting with `someapp1` so that the number is always part of the application name.

Looking forward

In the next chapter, we'll expand some of these top-level design ideas and look at module and package design. A small Python application can also be a module, which means it can be imported into a larger application. A complex Python application may be a package. It may include other application modules and it may be included in larger-scale applications.

Module and Package Design

Python gives us several constructs to group and organize our software. In `Section 1`, *Tighter Integration via Special Methods*, we looked at several techniques for using a class definition to combine data structures and behavior together, and create discrete objects defined by structure and behavior. In this chapter, we'll look at modules to encapsulate class and function definitions, as well as shared objects. We'll also look at packages as a design pattern to group related modules together.

Python makes it very easy to create simple modules. Any time we create a Python file, we're creating a module. As the scope of our designs gets larger and more sophisticated, the use of packages becomes more important to maintain a clear organization among the modules. This chapter will suggest patterns for module definition.

Some languages encourage putting a single class in a single file; this rule doesn't apply to Python. The Pythonic practice is to treat a whole module as a unit of reuse; it's common practice to have many closely-related class and function definitions in a single module.

Python has some specialized, reserved module names. For a larger application, we may implement a `__main__` module. This module must be designed to expose the OS command-line interface for a complex application. We also have some flexibility in how we install the modules associated with an application. We can use the default working directory, an environment variable setting, or the Python `lib/site-packages` directory. Each of these approaches has advantages and disadvantages.

Technical requirements

The code files for this chapter are available at https://git.io/fj2US.

One common approach is a single module. The overall organization can be imagined this way:

```
module.py
 ┣━━ class A:
 ┃    ┗━━ def method(self): ...
 ┣━━ class B:
 ┃    ┗━━ def method(self): ...
 ┗━━ def function(): ...
```

This example shows a single module with a number of classes and functions. We'll turn to module design in the *Designing a module* section, later in this chapter.

A more complex approach is a package of modules, which can be imagined as follows:

```
package
 ┣━━ __init__.py
 ┣━━ module1.py
 ┃    ┣━━ class A:
 ┃    ┃    ┗━━ def method(self): ...
 ┃    ┗━━ def function(): ...
 ┗━━ module2.py
 ┗━━ ...
```

This example shows a single package with two modules. Each module contains classes and functions. We'll look at package design in the *Designing a package* section later in this chapter.

We're going to avoid the more complex problem of distributing Python code. There are a number of techniques to create a source distribution for a Python project. The various distribution technologies are outside the scope of this book. The *Software Packaging and Distribution* section of the *Python Standard Library* addresses some of the physical file packaging issues. The *Distributing Python Modules* document provides information on creating a code distribution.

In this chapter, we will cover the following topics:

- Designing a module
- Whole modules versus module items
- Designing a package
- Designing a main script and the *main* module
- Designing long-running applications
- Organizing code into `src`, `scripts`, `tests`, and `docs`
- Installing Python modules

Designing a module

In chapters two through nine of this book, we looked at a number of techniques for designing classes, which are the foundation of object-oriented design and programming. The module is a collection of classes; it is a higher-level grouping of related classes and functions. It's rare to try to reuse a single class in isolation.

Consequently, the module is a fundamental component of Python implementation and reuse. A properly designed module can be reused because the needed classes and functions are bundled together. All Python programming is provided at the module level.

A Python module is a file. The filename extension must be `.py`. The filename in front of `.py` must be a valid Python name. Section 2.3 of *Python Language Reference* provides us with the complete definition of a name. One of the clauses in this definition is as follows:

"Within the ASCII range (U+0001..U+007F), the valid characters for identifiers are the uppercase and lowercase letters A through Z, the underscore _ and, except for the first character, the digits 0 through 9."

Operating system (**OS**) filenames permit more characters from the ASCII range than Python names; this extra OS complexity must be avoided. In particular, hyphens are a potential problem in Python module names; use underscores instead in complex file names. Because the stem of the filename (without the `.py` extension) becomes the module name, these names should also be valid Python identifiers.

TIP *The Python runtime may also create additional `.pyc` and `.pyo` files for its own private purposes; it's best to simply ignore these files. Generally, they're cached copies of code objects used to reduce the time to load a module. These files should be ignored.*

Every time we create a `.py` file, we create a module. Often, we'll create Python files without doing much design work. This simplicity is a benefit of using Python. In this chapter, we'll take a look at some of the design considerations to create a reusable module.

Now let's take a look at some of the design patterns for Python modules.

Some module design patterns

There are three commonly seen design patterns for Python modules:

- **Importable library modules**: These are meant to be imported. They contain definitions of classes, functions, and perhaps some assignment statements to create a few global variables. They do not do any real work; they can be imported without any worry about the side effects of the import operation. There are two use cases that we'll look at:
  - **Whole module**: Some modules are designed to be imported as a whole, creating a module namespace that contains all of the items.
  - **Item collection**: Some modules are designed to allow individual items to be imported, instead of creating a module object; the `math` module is a prime example of this design.
- **Runnable script modules**: These are meant to be executed from the command line. They contain more than class and function definitions. A script will include statements to do the real work. The presence of side effects means they cannot be meaningfully imported.
- **Conditional Script modules**: These modules are hybrids of the two aforementioned use cases: they can be imported and they can also be run from the command line. These modules will have the main-import switch as described in the *Python Standard Library*, in the *__main__ – Top-level script environment* section.

Here's the conditional script switch from the library documentation:

```
if __name__ == "__main__":
    main()
```

This requires a `main()` function to do the work of the script. This design supports two use cases: both **runnable** and **importable**. When the module is run from the command line, it evaluates `main()` and does the expected work. When the module is imported, the function will not be evaluated, and the import will simply create the various definitions without doing any real work.

We suggest something a bit more sophisticated, as shown in `Chapter 18`, *Coping with the Command Line*:

```
if __name__ == "__main__":
    with Setup_Logging():
        with Build_Config() as config:
            main = Simulate_Command()
            main.configure(config)
            main.run()
```

This leads to the following essential design tip:

> TIP: *Importing a module should have few side effects.*

Creating a few module-level variables is an acceptable side effect of an import. The real work – accessing network resources, printing output, updating files, and other kinds of processing – should not happen when a module is being imported.

A main script module without a `__name__ == "__main__"` section is often a bad idea because it can't be imported and reused. Beyond that, it's difficult for documentation tools to work with a main script module, and it's difficult to test. The documentation tools tend to import modules, causing work to be done unexpectedly. Similarly, testing requires care to avoid importing the module as part of a test setup.

In the next section, we'll compare a module with a class definition. The two concepts are similar in many respects.

Modules compared with classes

There are numerous parallels between the definitions for modules and classes:

- Both modules and classes have names defined by the Python syntax rules. To help distinguish them, modules usually have a leading lowercase letter; classes usually have a leading uppercase letter.
- Module and class definitions are namespaces that contain other objects.
- A module is a **Singleton** object within a global namespace, `sys.modules`. A class definition is unique within a namespace, either the global namespace, `__main__`, or some local namespace. A class definition is slightly different from a module **Singleton**, because a class definition can be replaced. Once imported, a module can't be imported again without using special functions such as `importlib.reload()`.
- The definitions of both class and module are evaluated as a sequence of statements within a namespace.
- A function defined in a module is analogous to a static method within a class definition.
- A class defined in a module is analogous to a class defined within another class.

There are two significant differences between a module and class:

- We can't create an instance of a module; it's always a **Singleton**. We can create multiple instances of a class.
- An assignment statement in a module creates a variable that's global within the module's namespace; it can be used by other functions in a module without using the module name as a qualifier for the namespace. An assignment statement within a class definition, however, creates a variable that's part of the class namespace: it requires a qualifier to distinguish it from the global variables outside the class.

TIP *Modules, packages, and classes can all be used to encapsulate data and process it into a tidy object. Classes can have multiple instances; modules cannot.*

The similarities between modules and classes mean that choosing between them is a design decision, with tradeoffs and alternatives. In most cases, the need for

multiple *instances* with distinct states is the deciding factor. Because a module is a **Singleton**, we cannot have discrete instances.

A module's **Singleton** pattern means that we'll use a module (or package) to contain class and function definitions. These definitions are then created exactly once, even if an `import` statement mentions them multiple times. This is a wonderful simplification, allowing us to repeat `import` statements in a variety of contexts.

The `logging` module, for example, is often imported in multiple other modules. The **Singleton** pattern means that the logging configuration can be done once, and will apply to all other modules. A configuration module, similarly, might be imported in several places. The singleton nature of a module ensures that the configuration can be imported by any module but will be truly global.

When writing applications that work with a single connected database, a module with a number of access functions will be similar to a singleton class. The database access layer can be imported throughout an application but will be a single, shared global object.

Once we've decided on the general design pattern for a module, the next consideration is to be sure it has all of the commonly-expected elements.

In the next section, we'll look at the expected content of a module.

The expected content of a module

Python modules have a typical organization. To an extent, this is defined by PEP 8, about which more information can be found at http://www.python.org/dev/peps/pep-0008/.

The first line of a module can be a `#!` comment; a typical version looks like the following code:

```
#!/usr/bin/env python3
```

This is used to help OS tools, such as `bash`, locate the Python interpreter for an executable script file. For Windows, this line may be something along the lines of `#!C:\Python3\python.exe`.

Older Python 2 modules may include a coding comment to specify the encoding for the rest of the text. This may look like the following code:

```
# -*- coding: utf-8 -*-
```

The coding comment is avoided for Python 3; the OS encoding information is adequate.

The next lines of a module should be a triple-quoted module docstring that defines the contents of the module file. As with other Python docstrings, the first paragraph of the text should be a summary. This should be followed by a more complete definition of the module's contents, purpose, and usage. This may include RST markup so that the documentation tools can produce elegant-looking results from the docstring. We'll address this in Chapter 20, *Quality and Documentation*.

After the docstring, we can include any version control information. For example, we might have the following code:

```
__version__ = "2.7.18"
```

This is a module global variable that we might use elsewhere in our application to determine the version number of the module. This is included after the

docstring, but before the body of the module. Below this, come all of the module's `import` statements. Conventionally, they're in a big block at the front of the module.

After the `import` statements, come the various class and function definitions of the module. These are presented in whatever order is required to ensure that they work correctly and make sense to someone who is reading the code.

TIP *Java and C++ tend to focus on one class per file. That's a silly limitation. It doesn't apply to Python.*

If the file has a lot of classes, we might find that the module is a bit hard to follow. If we find ourselves using big comment billboards to break a module into sections, this is a hint that what we're writing may be more complex than a single module.

A billboard comment looks like the following example:

```
################################
# FUNCTIONS RELATED TO API USE #
################################
```

Rather than use a billboard-style comment, it's better to decompose the module into separate modules. The billboard comment should become the docstring for a new module. In some cases, class definitions might be a good idea for decomposing a complex module.

Sometimes, global variables within a module are handy. The `logging` module makes use of this to keep track of all of loggers that an application might create. Another example is the way the `random` module creates a default instance of the `Random` class. This allows a number of module-level functions to provide a simple API for random numbers. We're not forced to create an instance of `random.Random`.

The PEP-8 conventions suggest these module globals should have `ALL_CAPS` style names to make them visible. Using a tool like `pylint` for code quality checks will result in this suggestion for global variables.

The next section compares whole modules with module items.

Whole modules versus module items

There are two approaches to designing the contents of a library module. Some modules are an integrated whole, while others are more like a collection of loosely related items. When we've designed a module as a whole, it will often have a few classes or functions that are the public-facing API of the module. When we've designed a module as a collection of loosely related items, each individual class or function tends to stand alone.

We often see this distinction in the way we import and use a module. We'll look at three variations:

- Using the `import some_module` command: This leads to the `some_module.py` module being evaluated and the resulting objects are collected into a single namespace called `some_module`. This requires us to use qualified names for all of the objects in the module, for example, `some_module.this` and `some_module.that`. This use of qualified names makes the module an integrated whole.
- Using the `from some_module import this, that` command: This leads to the `some_module.py` module file being evaluated and only the named objects are created in the current local namespace. We can now use `this` or `that` without the module namespace as a qualifier. This use of unqualified names is why a module can seem like a collection of disjointed objects. A common example is a statement like `from math import sqrt, sin, cos` to import a few math functions.
- Using the `from some_module import *` command: This will import the module and make all non-private names part of the namespace performing the import. A private name begins with `_`, and will not be retained as one of the imported names. We can explicitly limit the number of names imported by a module by providing an `__all__` list within the module. This list of string object names will be elaborated by the `import *` statement. We often use the `__all__` variable to conceal the utility functions that are part of building the module, but not part of the API that's provided to clients of the module.

When we look back at our design for decks of cards, we could elect to keep the suits as an implementation detail that's not imported by default. If we had a

`cards.py` module, we could include the following code:

```
from enum import Enum

__all__ = ["Deck", "Shoe"]

class Suit(str, Enum):
    Club = "\N{BLACK CLUB SUIT}"
    Diamond = "\N{BLACK DIAMOND SUIT}"
    Heart = "\N{BLACK HEART SUIT}"
    Spade = "\N{BLACK SPADE SUIT}"

class Card: ...

def card(rank: int, suit: Suit) -> Card: ...

class Deck: ...

class Shoe(Deck): ...
```

The use of the `__all__` variable makes the `Suit` and `Card` names visible. The `card()` function, the `Suit` class, and the `Deck` class are implementation details not imported by default. For example, suppose we perform the following code:

```
from cards import *
```

The preceding statement will only create `Deck` and `Shoe` in an application script, as those are the only explicitly given names in the `__all__` variable.

When we execute the following command, it will import the module without putting any names into the global namespace:

```
import cards
```

Even though it's not imported into the namespace, we can still access the qualified `cards.card()` method to create a `Card` instance.

There are advantages and disadvantages of each technique. A whole module requires using the module name as a qualifier; this makes the origin of an object explicit. Importing items from a module shortens their names, which can make complex programming more compact and easier to understand.

In the next section, we'll see how to design a package.

Designing a package

One important consideration when designing a package is *don't*. The *Zen of Python* poem (also known as `import this`) includes this line:

"Flat is better than nested"

We can see this in the Python standard library. The structure of the library is relatively flat; there are few nested modules. Deeply nested packages can be overused. We should be skeptical of excessive nesting.

A Python package is a directory with an extra file, `__init__.py`. The directory name must be a proper Python name. OS names include a lot of characters that are not allowed in Python names.

We often see three design patterns for packages:

- Simple packages are a directory with an empty `__init__.py` file. This package name becomes a qualifier for a collection of modules inside the package. We'll use the following code to pick one of the modules from the package:

```
import package.module
```

- A module-package hybrid can have an `__init__.py` file that is effectively a module definition. This top-level module will import elements from modules inside the package, exposing them via the `__init__` module. We'll use the following code to import the whole package as if it was a single module:

```
import package
```

- Another variation on the module-package hybrid uses the `__init__.py` file to choose among alternative implementations. We use the package as if it was a single module via code, as in the following example:

```
import package
```

The first kind of package is relatively simple. We add an `__init__.py` file to a directory, and, with that, we're done creating a package. The other two are a bit

more involved; we'll look at these in detail.

Let's see how to design a module-package hybrid.

Designing a module-package hybrid

In some cases, a design evolves into a module that is very complex; it can become so complex that a single file becomes a bad idea. When we start putting billboard comments in a module, it's a hint that we should consider refactoring a complex module into a package built from several smaller modules.

In this case, the package can be as simple as the following kind of structure. We can create a directory, named `blackjack`; within this directory, the `__init__.py` file would look like the following example:

```
"""Blackjack package"""
from blackjack.cards import Shoe
from blackjack.player import Strategy_1, Strategy_2
from blackjack.casino import ReSplit, NoReSplit, NoReSplitAces,  Hit17, Stand17
from blackjack.simulator import Table, Player, Simulate
from betting import Flat, Martingale, OneThreeTwoSix
```

This shows us how we can build a module-like package that is actually an assembly of parts imported from subsidiary modules. An overall application could be named `simulate.py`, containing code that looks like the following:

```
from blackjack import *
table = Table(
    decks=6, limit=500, dealer=Hit17(), split=NoReSplitAces(),
    payout=(3,2))
player = Player(
    play=Strategy_1(), betting=Martingale(), rounds=100,
    stake=100)
simulate = Simulate(table, player, 100)
for result in simulate:
    print(result)
```

This snippet shows us how we can use `from blackjack import *` to create a number of class definitions that originate in a number of other modules within the `blackjack` package. Specifically, there's an overall `blackjack` package that has the following modules within it:

- The `blackjack.cards` package contains the `Card`, `Deck`, and `Shoe` definitions.
- The `blackjack.player` package contains various strategies for play.
- The `blackjack.casino` package contains a number of classes that customize how casino rules vary.
- The `blackjack.simulator` package contains the top-level simulation tools.

- The `betting` package is also used by the application to define various betting strategies that are not unique to Blackjack but apply to any casino game.

The architecture of this package may simplify how we upgrade or extend our design. If each module is smaller and more focused, it's more readable and more understandable. It may be simpler to update each module in isolation.

Let's see how to design a package with alternate implementations.

Designing a package with alternate implementations

In some cases, we'll have a top-level `__init__.py` file that chooses between some alternative implementations within the package directory. The decision might be based on the platform, CPU architecture, or the availability of OS libraries.

There are two common design patterns and one less common design pattern for packages with alternative implementations:

- Examine `platform` or `sys` to determine the details of the implementation and decide what to import with an `if` statement.
- Attempt `import` and use a `try` block exception handling to work out the configuration details.
- As a less common alternative, an application may examine a configuration parameter to determine what should be imported. This is a bit more complex. We have an ordering issue between importing an application configuration and importing other application modules based on the configuration. It's far simpler to import without this potentially complex sequence of steps.

We'll show the structure of a hypothetical package named `some_algorithm`. This will be the name of the top-level directory. To create the complex package, the `some_algorithm` directory must include a number of files, described as follows:

- An `__init__.py` module will decide which of the two implementations to import. This name is required to define a package. The contents of this module will be shown in the following code block.
- An `abstraction.py` can provide any necessary abstract definitions for the two implementations. Using a single, common module can help to provided consistent type hints for `mypy` checking.
- Each implementation will be another module in the package. We'll outline two implementation choices, called `short_module.py` and `long_module.py`. Neither of these module names will be visible outside the package.

Here's `__init__.py` for a `some_algorithm` package. This module chooses an implementation based on the platform information. This might look like the following example:

```
import sys
from typing import Type

from Chapter_19.some_algorithm.abstraction import AbstractSomeAlgorithm

SomeAlgorithm: Type[AbstractSomeAlgorithm]

if sys.platform.endswith("32"):
    from Chapter_19.some_algorithm.short_version import *
    SomeAlgorithm = Implementation_Short
else:
    from Chapter_19.some_algorithm.long_version import *
    SomeAlgorithm = Implementation_Long
```

This module defines the `SomeAlgorithm` class based on one of two available implementation modules. For 32-bit platforms, the `short_version.py` module provides a class named `Implementation_Short` that will be used. For 64-bit platforms, the `long_version.py` module provides the `Implementation_Long` class.

We need to also provide two modules within the `some_algorithm` package; the `long_version.py` module provides an implementation appropriate for a 64-bit architecture; the `short_version` module provides an alternate implementation. The design must have module isomorphism; this is similar to class isomorphism. Both the modules must contain classes and functions with the same names and same APIs.

If both the files define a class named `SomeClass`, then we can write the following code in an application:

```
from Chapter_19 import some_algorithm
x = some_algorithm.SomeAlgorithm()
```

We can import the `some_algorithm` package as if it were a module. This will import the `some_algorithm/__init__.py` module. This module locates an appropriate implementation and provides the needed class definition.

Each implementation is similar. Both will incorporate the abstract class to make it clear to tools such as `mypy` that the two implementations are identical. Here is the content of the `short_implementation.py` module.

```
from .abstraction import AbstractSomeAlgorithm
```

```
class Implementation_Short(AbstractSomeAlgorithm):

    def value(self) -> int:
        return 42
```

This module imports an abstract class definition. It then defines a proper subclass. This overhead helps `mypy` confirm the defined class is a complete implementation of the abstract class definition.

For complex applications, this kind of alternative implementation strategy can be very helpful. It lets a single code base work in a number of environments where configuration changes are made as late in the deployment pipelines as possible.

The next section shows how to use the `ImportError` exception.

Using the ImportError exception

An alternative to an `if` statement is to use a `try` statement to locate a candidate's implementation. This technique works well when there are different distributions. Often, a platform-specific distribution may include files that are unique to the platform.

In Chapter 16, *The Logging and Warning Modules*, we showed you this design pattern in the context of providing warnings in the event of a configuration error or problem. In some cases, tracking down variant configurations doesn't deserve a warning, because the variant configuration is a design feature.

Here's `__init__.py` for a `some_algorithm` package, which chooses an implementation based on the availability of the module files within the package:

```
try:
    from some_algorithm.long_version import *
except ImportError as e:
    from some_algorithm.short_version import *
```

This depends on having two distinct distributions that will include either the `some_algorithm/long_version.py` file or the `some_algorithm/short_version.py` file. If the `some_algorithm.long_version` module is not found, then `some_alogirithm.short_version` will be imported. The content of the implementation modules doesn't change from what was shown in the preceding code block. Only the `__init__.py` module will change.

Creating variant distributions is outside the scope of this book. The Python Packaging Authority, PyPA, has documentation showing how platform-specific wheel and egg files can be created.

This try/except technique doesn't scale to more than two or three alternative implementations. As the number of choices grows, the `except` blocks will become very deeply nested.

Let's see how to design a main script and the *main* module.

Designing a main script and the __main__ module

A top-level main script will execute our application. In some cases, we may have multiple main scripts because our application does several things. We have three general approaches to writing the top-level main script:

- For very small applications, we can run the application with `python3 some_script.py`. This is the style that we've shown you in most examples.
- For some larger applications, we'll have one or more files that we mark as executable with the OS `chmod +x` command. We can put these executable files into Python's `scripts` directory with our `setup.py` installation. We run these applications with `some_script.py` at the command line.
- For complex applications, we might add a `__main__.py` module in the application's package. To provide a tidy interface, the standard library offers the `runpy` module and the `-m` command-line option that will use this specially named module. We can run this with `python3 -m some_app`.

We'll look at the last two options in detail.

Creating an executable script file

To use an executable script file, we have a two-step implementation: make it executable and include a `#!` (sharp-bang, or *shebang*) line. We will take a look at the details.

The Linux command to mark a script file as executable looks like the following example:

```
chmod +x some_script.py
```

The shebang line will often look like this example:

```
#!/usr/bin/env python3
```

This line will direct the OS to use the named program to execute the script file. In this case, we used the `/usr/bin/env` program to locate the `python3` program to run the script. The `python3` program will be given the script file as its input.

When a script file is marked executable, and the file includes the `#!` line, we can use `some_script.py` at the command line to run the script.

For a more complex application, this top-level script may import other modules and packages. It's important that these top-level executable script files should be as simple as possible to promote reuse of the various components. Key design principles include the following:

- Keep the script module as small as possible. Any complexity should be in the modules which are imported.
- A script module should have no new or distinctive code. It should emphasize importing and using code from other modules.
- In the long run, no program will ever stand alone. Any valuable software will be extended and repurposed. Even the top-level script for an application may be integrated into an even larger wrapper.

Our design goals must always include the idea of composite, larger-scale programming. A main script file should be as short as possible. Here's our

example:

```
import simulation

if __name__ == "__main__":
    with simulation.Setup_Logging():
        with simulation.Build_Config() as config:
            main = simulation.Simulate_Command()
            main.configure(config)
            main.run()
```

All of the relevant working code is imported from a module named `simulation`. There's no unique or distinctive new code introduced in this module.

In the next section, we'll see how to create a `_main_` module .

Creating a __main__ module

To work with the `runpy` interface, we must add a small `__main__.py` module to our application's top-level package. We have emphasized the design of this top-level executable script file.

We should always permit refactoring an application to build a larger, more sophisticated composite application. If there's functionality buried in `__main__.py`, we need to pull this into a module with a clear, importable name so that it can be used by other applications.

A `__main__.py` module follows the kind of code shown in the previous section, *Creating an executable script file*. The only real distinction is using the special name, `__main__.py`, to make it easier for the Python runtime to locate the main module of a package. It also makes it easier for other people to locate the main part of a package's processing.

One of the important considerations in Python programming is combining multiple, smaller programs into useful, larger programs. In the next section, we'll look at aggregation, or *programming in the large*.

Programming in the large

Here's an example that shows us why we shouldn't put unique, working code into the `__main__.py` module. We'll show you a quick hypothetical example based on extending an existing package.

Imagine that we have a generic statistical package, named `analysis`, with a top-level `__main__.py` module. This implements a command-line interface that will compute descriptive statistics of a given CSV file. This application has a command-line API as follows:

```
python3 -m analysis -c 10 some_file.csv
```

This command uses a `-c` option to specify which column to analyze. The input filename is provided as a positional argument on the command line.

Let's assume, further, that we have a terrible design problem. We've defined a high-level function, `analyze()`, in the `analysis/__main__.py` module. Here's an outline of the `__main__.py` module:

```
import argparse
from analysis import some_algorithm

def analyze(config: argparse.Namespace) -> None: ...

def main(argv: List[str] = sys.argv[1:]) -> None: ...

if __name__ == "__main__":
    main(sys.argv[1:])
```

The `analysis` package includes a `__main__.py` module. This module does more than just simply run functions and classes defined elsewhere. It also includes a unique, reusable function definition, `analyze()`. This is not a problem until we try to reuse elements of the `analysis` package.

Our goal is to combine this with our Blackjack simulation. Because of the design error here, this won't work out well. We might *think* we can do this:

```
import analysis
import simulation
import types
```

```
def sim_and_analyze():
    with simulation.Build_Config() as config_sim:
        config_sim.outputfile = "some_file.csv"
        s = simulation.Simulate()
        s.configure(config_sim)
        s.run()
    config_stats = types.SimpleNamespace(
        column=10, input="some_file.csv")
    analysis.analyze(config_stats)
```

We tried to use `analysis.analyze()`, assuming that the useful `analyze()` function was part of a simple module. The Python naming rules make it appear as though `analysis` is a module with a function named `analyze()`. Most of the time, the implementation details of the module and package structure don't matter for successful use. This, however, is an example of reuse and *Programming In The Large*, where the structure needs to be transparent.

This kind of simple composition was made needlessly difficult by defining a function in `__main__`. We want to avoid being forced to do this:

```
def analyze(column, filename):
    import subprocess
    subprocess.run(
        ["python3", "-m", "stats", "-c", column, filename])
```

We shouldn't need to create composite Python applications via the command-line API. In order to create a sensible composition of the existing applications, we might be forced to refactor `analysis/__main__.py` to remove any definitions from this module and push them up into the package as a whole.

The next section shows how to design long-running applications.

Designing long-running applications

A long-running application server will be reading requests from some kind of queue and formulating responses to those requests. In many cases, we leverage the HTTP protocol and build application servers into a web server framework. See `Chapter 13`, *Transmitting and Sharing Objects*, for details on how to implement RESTful web services following the **Web Server Gateway Interface** (**WSGI**) design pattern.

A desktop GUI application has a lot of features in common with a server. It reads events from a queue that includes mouse and keyboard actions. It handles each event and gives some kind of GUI response. In some cases, the response may be a small update to a text widget. In other cases, a file might get opened or closed, and the state of menu items may change.

In both cases, the central feature of the application is a loop that runs forever, handling events or requests. Because these loops are simple, they're often part of the framework. For a GUI application, we might have a loop like that in the following code:

```
root = Tkinter.Tk()
app = Application(root)
root.mainloop()
```

For `Tkinter` applications, the top-level widget's `mainloop()` gets each GUI event and hands it to the appropriate framework component for handling. When the object handling events—the top-level widget, `root`, in the example—executes the `quit()` method, then the loop will be gracefully terminated.

For a WSGI-based web server framework, we might have a loop like the following code:

```
httpd = make_server('', 8080, debug)
httpd.serve_forever()
```

In this case, the server's `serve_forever()` method gets each request and hands it to the application—`debug` in this example—for handling. When the application executes the server's `shutdown()` method, the loop will be gracefully terminated.

We often have some additional requirements that distinguish long-running applications:

- **Robust**: When dealing with external OS or network resources, there are timeouts and other errors that must be confronted successfully. An application framework that allows for plugins and extensions, enjoys the possibility of an extension component harboring an error that the overall framework must handle gracefully. Python's ordinary exception handling is perfectly adequate for writing robust servers. In `Chapter 15`, *Design Principles and Patterns*, we addressed some of the high-level considerations.
- **Auditable**: A simple, centralized log is not always sufficient. In `Chapter 16`, *The Logging and Warning Modules*, we addressed techniques to create multiple logs to support the security or financial audit requirements.
- **Debuggable**: Ordinary unit testing and integration testing reduces the need for complex debugging tools. However, external resources and software plugins or extensions create complexities that may be difficult to handle without providing some debugging support. More sophisticated logging can be helpful.
- **Configurable**: Except for simple technology spikes, we want to be able to enable or disable the application features. Enabling or disabling debugging logs, for example, is a common configuration change. In some cases, we want to make these changes without completely stopping and restarting an application. In `Chapter 14`, *Configuration Files and Persistence*, we looked at some techniques to configure an application. In `Chapter 18`, *Coping with the Command Line*, we extended these techniques.
- **Controllable**. A simplistic long-running server can simply be killed in order to restart it with a different configuration. In order to ensure that buffers are flushed properly and OS resources are released properly, it's better to use a signal other than `SIGKILL` to force termination. Python has signal-handling capabilities available in the `signal` module.

These last two requirements for a dynamic configuration and clean way to shut down a server, lead us to separate the primary input stream from a secondary control input. This control input can provide additional requests for configuration or shutdown.

We have a number of ways to provide asynchronous inputs through an additional channel:

- One of the simplest ways is to create a queue using the `multiprocessing` module. In this case, a simple administrative client can interact with this queue to control or interrogate the server or GUI. For more examples of `multiprocessing`, see `Chapter 13`, *Transmitting and Sharing Objects*. We can transmit the control or status objects between the administrative client and the server.
- Lower-level techniques are defined in the *Networking and Interprocess Communication* section of the *Python Standard Library*. These modules can also be used to coordinate with a long-running server or GUI application.
- Use persistent storage of state so the long-running process can be killed and restarted with a different configuration. We looked at persistence techniques in `Chapter 10`, *Serializing and Saving - JSON, YAML Pickle, CSV and Shelve;* `Chapter 11`, *Storing and Retrieving Objects via Shelve;* and `Chapter 12`, *Storing and Retrieving Objects via SQLite*. Any of these can be used to save the state of a server leading to a seamless restart.

There are two common use cases for web-based servers:

- Some servers provide a RESTful API.
- Some servers are focused on providing the **User Experience** (**UX**).

The RESTful API servers are often used by mobile applications, and the UX is packaged separately. A RESTful API server generally maintains state in a persistent database. As part of server reliability engineering, there may be multiple copies of the server to share the workload, and software upgrades often happen by introducing the new release and shifting the workload from the old servers to the new servers. When there are multiple copies of a server, then shared persistent storage is required to keep the state of a user's transaction when each individual request could be handled by different servers. The dynamic configuration and control is handled by shifting workloads and stopping the old servers so the work is handled by new servers.

Providing a UX from a web server often requires maintaining session state on the server. In this case, killing the server means the user's session state is lost. We don't want to have angry users losing the contents of their shopping carts because we reconfigured a server. If the session information is cached in a database, and the cookie sent to the user is nothing more than a database key, we can create very robust web servers. The FlaskSession project (more info

available at `https://pythonhosted.org/Flask-Session`) provides a number of ways of saving session information in a cache so servers can be stopped and restarted.

The next section shows how to organize code into `src`, `scripts`, `tests`, and `docs`.

Organizing code into src, scripts, tests, and docs

As we noted in the previous section, there's no essential need for a complex directory structure in a Python project. The ideal structure follows the standard library, and is a relatively flat list of modules. This will include overheads such as the `setup.py` and `README` files. This is pleasantly simple and easy to work with.

When the modules and packages get more complex, we'll often need to impose some structure. For complex applications, one common approach is to segregate Python code into a few bundles. To make the examples concrete, let's assume that our application is called `my_app`:

- `my_app/src`: This directory has all of the working application code. All of the various modules and packages are here. In some cases, there's only a single top-level package name in this `src` directory. In other cases, there may be many modules or packages listed under `src`. The advantage of a `src` directory is the simplicity of doing static analysis with `mypy`, `pylint`, or `pyflakes`.
- `my_app/scripts` or `my_app/bin`: This directory can have any scripts that form an OS-level command-line API. These scripts can be copied to the Python `scripts` directory by `setup.py`. As noted previously, these should be like the `__main__.py` module; they should be very short, and they can be thought of as OS filename aliases for Python code.
- `my_app/tests`: This directory can have the various test modules. Most of the modules will have names beginning with `test_` so they can be discovered by `pytest` automatically.
- `my_app/docs`: This directory will have the documentation. We'll look at this in `Chapter 20`, *Quality and Documentation*.

The top-level directory name, `my_app`, might be augmented with a version number to permit having multiple branches or versions available, leading to `my_app-v1.1` as a top-level directory name. A better strategy is to use a sophisticated version control tool, such as **git**. This can manage having multiple versions of software in a single directory structure. Using **git** commands to switch branches works out better than trying to have multiple branches in adjacent directories.

The top-level directory will contain the `setup.py` file to install the application into Python's standard library structure. See *Distributing Python Modules* (https://docs.python.org/3/distributing/index.html) for more information. Additionally, of course, a `README.rst` file would be placed in this directory. Other common files found here are the `tox.ini` file, for configuring the overall testing environment, and the `environment.yaml` file, for building the Conda environment for using the application.

When the application modules and test modules are in separate directories, we need to refer to the application as an installed module when running tests. We can use the `PYTHONPATH` environment variable for this. We can run the test suite as in the following code:

```
PYTHONPATH=my_app/src python3 -m test
```

We set an environment variable on the same line where we execute a command. This may be surprising, but it's a first-class feature of the `bash` shell. This allows us to make a very localized override to the `PYTHONPATH` environment variable.

The next section shows how to install Python modules.

Installing Python modules

We have several techniques to install a Python module or package:

- We can write `setup.py` and use the distribution utilities module, `distutils`, to install the package into Python's `lib/site-packages` directory. This is described in detail in the Python Packaging Authority documentation. See `https://www.pypa.io/en/latest/` for information. Building software for other people to install can be complex, and will often require sophisticated test cases using the **tox** tool to build environments, run tests, and create distribution kits. See `https://tox.readthedocs.io/en/latest/` for more information.
- We can set the `PYTHONPATH` environment variable to include our packages and modules. We can set this temporarily in a shell, or we can set it more permanently by editing our `~/.bash_profile` or the system's `/etc/profile`. We'll take a look at this later in this section.
- The current working directory is a package as well. It's always first on the `sys.path` list. When working on a simple one-module Python application, this is very handy.

Setting the environment variable can be done transiently or persistently. We can set it in an interactive session with a command, as follows:

```
export PYTHONPATH=~/my_app-v1.2/src
```

This sets `PYTHONPATH` to include the named directory when searching for a module. The module is effectively installed through this simple change to the environment. Nothing is written to Python's `lib/site-packages`.

This is a transient setting that may be lost when we end the terminal session. The alternative is to update our `~/.bash_profile` to include a more permanent change to the environment. We simply append that `export` line to `.bash_profile` so that the package is used every time we log in.

For web server applications, the Apache, NGINX, or uWSGI configuration may need to be updated to include access to the necessary Python modules. There are two approaches to creating web servers:

- **Install Everything**. Use Python package installation to create the entire web server. This will involve using a local package index for customized application components. More work is done in the integration phase of continuous integration/continuous deployment (CI/CD).
- **Install Open Source Only**. Use Python package installation for open source components. Use Git checkout and `PYTHONPATH` for the customized application components. More work is done in the deployment phase of CI/CD.

Either approach will work nicely. While the *install everything* approach has the superficial advantage of simplicity, it leads to creating installable versions of all of our customized application software. For software that is proprietary, and will not be released as open source, this seems like needless integration work to create an installable version.

Installing only open source components leads to a more complex application deployment. There will be a `conda` (or `pip`) installation of open source components, in addition to **git** checkout of proprietary components.

Summary

We looked at a number of considerations to design modules and packages. The parallels between a module and singleton class are deep. When we design a module, the essential questions of the encapsulation of the structure and processing are as relevant as they are for class design.

When we design a package, we need to be skeptical of the need for deeply nested structures. We'll need to use packages when there are varying implementations; we looked at a number of ways to handle this variability. We may also need to define a package to combine a number of modules into a single module-like package. We looked at how `__init__.py` can import from within the package.

Design considerations and tradeoffs

We have a deep hierarchy of packaging techniques. We can simply organize the functionality into defined functions. We can combine the defined functions and their related data into a class. We can then combine related classes into a module. Lastly, we can combine related modules into a package.

When we think of software as a language to capture knowledge and representation, we have to consider what a class or module *means*. A module is the unit of the Python software construction, distribution, use, and reuse. With rare exceptions, modules must be designed around the possibility of reuse.

In most cases, we'll use a class, because we expect to have multiple objects that are instances of the class. Often – but not always – a class will have stateful instance variables.

When we look at classes with only a single instance, it's not perfectly clear if a class is truly necessary. Standalone functions may be as meaningful as a single-instance class. In some instances, a module of separate functions may be an appropriate design, because modules are inherently singletons.

The general expectation is a simple stateful collection of definitions. A module is a namespace that can also contain some local variables. This parallels a class definition, but lacks the ability to create instances.

While we can create immutable classes (using `__slots__`, extending `NamedTuple`, using a frozen `@dataclass`, or overriding the attribute setter methods), we can't easily create an immutable module. There doesn't seem to be a use case for an immutable module object.

A small application may be a single module. A larger application will often be a package. As with module design, packages should be designed for reuse. A larger application package should properly include a `__main__` module.

Looking forward

In the next chapter, we'll consolidate a number of our object-oriented design techniques. We'll take a look at the overall quality of our design and implementation. One consideration is assuring others that our software is trustworthy. One aspect of trustworthy software is coherent, easy-to-use documentation.

Quality and Documentation

To be valuable, software must be trusted. The general goal of trustworthiness relies on a number of software quality attributes. For more information on this, refer to the *S-Cube Quality Reference Model* section in `https://s-cube-network.eu/km/qrm/index.html`. Good documentation is an underlying technique for demonstrating that the various quality attributes have been met.

In this chapter, we'll look at two tools to produce the documentation from the code: `pydoc` and `Sphinx`. The `pydoc` tool extracts documentation from the Python code and produces useful views of the docstrings. The `Sphinx` tool allows us to create complete and sophisticated documentation using a lightweight markup language coupled with the source code. We'll describe some features of **reStructuredText** (**RST**) to help make our documentation more readable.

For more information, refer to PEP 257 in `https://www.python.org/dev/peps/pep-0257/`. This describes the minimum standard for docstrings. The use of formatting markup such as RST extends this baseline proposal.

Complete test cases are another aspect of high-quality software. We can use the `doctest` module to execute test cases. Doing this addresses two quality aspects—documentation and testing—with a single tool.

We'll also take a brief look at literate programming techniques. The idea is to write a pleasant, easy-to-understand document that contains the entire body of the source code along with explanatory notes and design details. Literate programming isn't simple, but it can produce good code coupled with a resulting document that is very clear and complete.

In this chapter, we will cover the following topics:

- Writing docstrings for the `help()` function
- Using pydoc for documentation
- Better output via RST markup
- Writing effective docstrings
- Writing file-level docstrings, including modules and packages

- More sophisticated markup techniques
- Using `Sphinx` to produce the documentation
- Writing the documentation
- Literate programming

Technical requirements

The code files for this chapter are available at `https://git.io/fj2U9`.

Writing docstrings for the help() function

Python provides numerous places to include the documentation. The definition of a package, module, class, or function has room for a string that includes a description of the object that is being defined.

Throughout this book, we avoided showing you lengthy docstrings in each example because our focus is on the Python programming details, not the overall software product that is being delivered.

As we move beyond advanced OO design and look at the overall deliverable product, docstrings become an important part of the deliverable. Docstrings can provide us with several key pieces of information:

- The API: the parameters, return values, and exceptions raised.
- A description of what to expect.
- Optionally, the `doctest` test results. For more information, refer to `Chapter 17`, *Designing for Testability*.

We can, of course, write even more in a docstring. We can provide more details on the design, architecture, and requirements. At some point, these more abstract, higher-level considerations are not directly tied to the Python code. This higher-level design and the requirements don't properly belong to the code or the docstrings.

The `help()` function extracts and displays the docstrings. It performs some minimal formatting on the text. The `help()` function is installed in the interactive Python environment by the `site` package. The function is actually defined in the `pydoc` package. In principle, we can import and extend this package to customize the `help()` output.

Writing documentation that is suitable for `help()` is relatively simple. Here's a typical example of output from `help(round)`:

```
Help on built-in function round in module builtins:
```

```
round(number, ndigits=None)
    Round a number to a given precision in decimal digits.

    The return value is an integer if ndigits is omitted or None. Otherwise
    the return value has the same type as the number. ndigits may be negative.
```

This shows us the required elements: the summary, the API, and the description. The API and the summary are the first line: `round(number, ndigits=None)`.

The description text defines what the function does. More complex functions may describe exceptions or edge cases that may be important or unique to this function. The `round()` function, for example, doesn't detail elements such as `TypeError` that might get raised.

A `help()`-oriented docstring is expected to be pure text with no markup. We can add some RST markup, but it isn't used by `help()`.

To make `help()` work, we simply provide docstrings. As it's so simple, there's no reason not to do it. Every function or class needs a docstring so that `help()` shows us something useful.

Now, let's see how to use pydoc for documentation.

Using pydoc for documentation

We use the library module, `pydoc`, to produce HTML documentation from Python code. It turns out that we're using it when we evaluate `help()` in interactive Python. This function produces the *text mode* documentation with no markup.

When we use `pydoc` to produce the documentation, we'll use it in one of the following ways:

- Prepare text-mode documentation files and view them with command-line tools such as `more` or `less`.
- Run an HTTP server and browse the documentation directly.

We can run the following command-line tool to prepare the text-based documentation for a module:

```
pydoc somemodule
```

We can also use the following code:

```
python3 -m pydoc somemodule
```

Either command will create text documentation based on the Python code. The output will be displayed with programs such as `less` (on Linux or macOS) or `more` (on Windows), which paginate the long stream of output.

Ordinarily, `pydoc` presumes that we're providing a module name to import. This means that the module must be on the Python path for ordinary import. As an alternative, we can specify a physical filename. Something along the lines of `pydoc ./mymodule.py` will work to pick a file instead of a module.

The program can also start a special-purpose web server to browse a package or module's documentation. In addition to simply starting the server, we can combine starting the server and launching our default browser. The `-b` option starts a browser. Here's a way to simply start a server and also launch a browser at the same time:

```
python3 -m pydoc -b
```

This will locate an unused port, start a server, and then launch your default browser to point at the server.

It's not easy to customize the output from `pydoc`. The various styles and colors are effectively hardcoded into the class definitions. Revising and expanding `pydoc` to use the external CSS styles would be an interesting exercise.

The following screenshot shows the default styling:

Classes

- builtins.object
  - Card
    - AceCard
    - FaceCard
- builtins.str(builtins.object)
  - Suit(builtins.str, enum.Enum)
- enum.Enum(builtins.object)
  - Suit(builtins.str, enum.Enum)

class **AceCard**(Card)

AceCard(rank: int, suit: src.ch20_ex1.Suit, hard: int, soft: Union[int, NoneType] = None) -> None

Subclass of :py:class:`Card` with rank of 1 represented by A.

Method resolution order:

- AceCard
- Card
- builtins.object

Methods defined here:

__str__(self) -> str

Return str(self).

Methods inherited from Card:

__init__(self, rank: int, suit: src.ch20_ex1.Suit, hard: int, soft: Union[int, NoneType] = None) -> None

Define the values for this card.

Now, let's see how to obtain a better output using RST markup.

Better output via RST markup

Our documentation could be much nicer if we use a more sophisticated toolset.

There are several things that we'd like to be able to do, such as the following:

- Fine-tune the presentation to include emphasis such as bold, italic, or color.
- Provide the semantic markup for the parameters, return values, exceptions, and cross-references among Python objects.
- Provide a link to view the source code.

- Filter the code that's included or rejected. This includes fine-tuning the presence of standard library modules, *private* objects with a leading _, system objects with a leading __, or superclass members.
- Adjust the CSS to provide a different style for the resulting HTML pages.

We can address the first two requirements through more sophisticated markup in our docstrings. We'll need to use the RST markup language, and we'll also need an additional tool to address the last three requirements.

Once we start using more sophisticated markup, we can branch out beyond HTML to include $\LaTeX$ for even better-looking documentation. This allows us to also produce the PostScript or PDF output in addition to HTML from a single source.

RST is a simple, lightweight markup. There are plenty of good tutorials and summaries associated with the Python `docutils` project. Refer to `http://docutils.sourceforge.net` for details.

A quick overview is available here: `http://docutils.sourceforge.net/docs/user/rst/quickstart.html`.

The point of the `docutils` toolset is that a very smart parser allows us to use very simple markup. HTML and XML rely on a relatively unsophisticated parser and put the burden on the human (or an editing tool) to create the complex markup. While XML and HTML allow for a wide variety of use cases, the `docutils` parser

is more narrowly focused on the natural language text. Because of the narrow focus, `docutils` is able to deduce our intent, based on the use of blank lines and some ASCII punctuation characters.

For our purposes, the `docutils` parser recognizes the following three fundamental things:

- Blocks of text: paragraphs, headings, lists, block quotes, code samples, and the `doctest` blocks. These are all separated by blank lines.
- Inline markup can appear inside the text blocks. This involves the use of simple punctuation to mark the characters within the text block. There are two kinds of inline markup; we'll look at the details in the later section.
- Directives are also blocks of text, but they begin with `..` as the first two characters of the line. Directives are open-ended and can be extended to add features to docutils.

The essential ingredient in writing text that can be transformed into good-looking documents is to clearly separate the blocks of text. In the next section, we'll look at the different kinds of text blocks.

Blocks of text

A block of text is simply a paragraph, set off from other paragraphs, by a blank line. This is the fundamental unit of the RST markup. RST recognizes a number of kinds of paragraphs, based on the pattern that is followed. Here's an example of a heading:

```
This Is A Heading
=================
```

This is recognized as a heading because it's *underlined* with a repeated string of special characters.

The `docutils` parser deduces the hierarchy of title underlines based entirely on their usage. We must be consistent with our headings and their nesting. It helps to pick a standard and stick to it. It also helps to keep documents fairly *flat* without complex, nested headings. Three levels are often all that's needed; this means that we can use `====`, `----`, and `~~~~` for the three levels.

A bullet list item begins with a special character; the content must also be indented. As Python uses a 4-space indent, this is common in RST as well. However, almost any consistent indent will work:

```
Bullet Lists

-   Leading Special Character.

-   Consistent Indent.
```

Note the blank line between paragraphs. For some kinds of simple bullet lists, the blank lines aren't required. In general, blank lines are a good idea.

A numerical list begins with a digit or letter and a roman numeral. To have numbers generated automatically, `#` can be used as the list item:

```
Number Lists

1.  Leading digit or letter.

#.  Auto-numbering with #.

#.  Looks like this.
```

We can use the indent rules to create lists within lists. It can be complex, and the `docutils` RST parser will usually figure out what you meant.

A block quote is simply indented text:

```
Here's a paragraph with a cool quote.

    Cool quotes might include a tip.

Here's another paragraph.
```

Code samples are indicated with a `::` double colon; they are indented, and they end with a blank line. While `::` can be at the end of a line or on a line by itself, putting `::` on a separate line makes it slightly easier to find code samples.

Here's a code sample:

```
Here's an example:
::

    x = Deck()
    first_card = x.pop()

This shows two lines of code. It will be distinguished from surrounding text.
```

The `docutils` parser will also locate the `doctest` material and set it aside for special formatting, similar to a code block. They begin with `>>>` and end with a blank line.

Here's some sample output from `doctest`:

```
Here's an example:
::

    >>> x = Unsorted_Deck()
    >>> x.pop()
    'A♣'

This shows how the :class:`Unsorted_Deck` class works.
```

A blank line at the end of the test output is essential and is easily overlooked. When the doctest error messages include the surrounding text, this means that additional blank lines are required.

One way in which we can annotate our text is by providing inline markup to identify things that should be emphasized in different ways. It might be code, or an important word, or a cross-reference. In the next section, we'll show how

inline markup is used in RST.

The RST inline markup

Within most blocks of text, we can include inline markup. We can't include inline markup in the code samples or `doctest` blocks. Note that we cannot nest inline markup, either.

The RST inline markup includes a variety of common ASCII treatments of text. For example, we have `*emphasis*` and `**strong emphasis**`, which will usually produce italic and bold, respectively. We may want to emphasize code segments within a block of text; we use ``` ``literal`` ``` to force a monospaced font.

We can also include cross-references as the inline markup. A trailing `_` indicates a reference, and it points away from the preceding words; a leading `_` indicates a target, and it points toward the following words. For example, we might have ``` `some phrase`_ ``` as a reference. We can then use ``` _`some phrase` ``` as the target for that reference. We don't need to provide explicit targets for section titles: we can reference ``` `This Is A Heading`_ ``` because all the section titles are already defined as targets. For the HTML output, this will generate the expected `<a>` tags. For the PDF output, in-text links will be generated.

We cannot nest inline markup. There's little need for nested inline markup; too many typographic tricks devolve to visual clutter. If our writing is so sensitive to typography, we should probably use LaTeX directly.

Inline markup can also have explicit role indicators; this is `:role:`, followed by `` `text` ``. Simple RST has relatively few roles. We might use ``:code:`some code` `` to be more explicit regarding the presence of a code sample in the text. When we look at Sphinx, there are numerous role indicators. The use of explicit roles can provide a great deal of semantic information.

When doing things that have more complex math, we might use the LaTeX math typesetting capabilities. This uses the `:math:` role; it looks like this: ``:math:`a=\pi r^2` ``.

Roles are open-ended. We can provide a configuration to docutils that adds new roles. This is done by means of tools such as Sphinx.

In addition to inline role definitions, RST has a number of directives. In the next section, we'll look at these directives.

RST directives

RST also includes directives. A directive is written in a block that starts with `..`. A directive may contain content that's indented. It may also have parameters. RST has a large number of directives that we might use to create a more sophisticated document. For docstring preparation, we'll rarely use more than a few of the directives available. The directives are open-ended; tools such as Sphinx will add directives to produce more sophisticated documentation.

Three commonly used directives are `image`, `csv-table`, and `math`. If we have an image that should be part of our document, we might include it in the following way:

```
..  image:: media/some_file.png
    :width: 6in
```

We named the file `media/some_file.png`. We also provided it with a `width` parameter to ensure that our image fits our document page layout. There are a number of other parameters that we can use to adjust the presentation of an image.

- `:align:`: We can provide keywords such as `top`, `middle`, `bottom`, `left`, `center`, or `right`. This value will be provided to the `align` attribute of the HTML `<img>` tag.
- `:alt:`: This is the alternative text for the image. This value will be provided to the `alt` attribute of the HTML `<img>` tag.
- `:height:`: This is the height of the image.
- `:scale:`: This is a scale factor that can be provided instead of the height and width.
- `:width:`: This is the width of the image.
- `:target:`: This is a target hyperlink for the image. This can be a complete URI or an RST reference of the `` `name`_ `` form.

For the height and width, any of the length units available in CSS can be used. These include `em` (the height of the element's font), `ex` (the height of the letter *x*), `px` (pixels), as well as absolute sizes; `in`, `cm`, `mm`, `pt` (point), and `pc` (pica).

We can include a table in our document in the following manner:

```
..  csv-table:: Suits
    :header: symbol, name

    "'♣'", Clubs
    "'♦'", Diamonds
    "'♥'", Hearts
    "'♠'", Spades
```

This allows us to prepare data that will become a complex HTML table in a simple CSV notation. We can have a more complex formula using the `math` directive:

```
..  math::

    \textbf{O}(2^n)
```

This allows us to write larger LaTeX math to create a separate equation. These can be numbered and cross-referenced as well. Note the blank lines and the indentation around the formula; these are essential for helping the RST tools locate the relevant text to be treated as math.

There are a number of other RST markup techniques. In the next section, we'll suggest some ways to learn how to annotate text so that it is properly formatted and indexed.

Learning RST

One way to build skills in RST is to install `docutils` and use the `rst2html.py` script to parse an RST document and convert it to HTML pages. A simple practice document can easily show us the various RST features.

All of a project's requirements, architecture, and documentation can be written using RST and transformed into HTML or LaTeX. It's relatively inexpensive to write user stories in RST and drop those files into a directory that can be organized and reorganized, since stories are groomed, put into development, and implemented. More complex tools may not be any more valuable than `docutils`.

The advantage of using pure text files and the RST markup is that we can easily manage our documentation in parallel with our source code. We're not using a proprietary word processing file format. We're not using a wordy and long-winded HTML or XML markup that must be compressed to be practical. We're simply storing more text along with the source code.

If we're using RST to create the documentation, we can also use the `rst2latex.py` script to create a `.tex` file that we can run through a LaTeX toolset to create postscript or PDF documents. This requires a LaTeX toolset, and usually, the **TeXLive** distribution is used for this. Refer to `http://www.tug.org/texlive/` for a comprehensive set of tools to transform TeX into elegant, final documents. TeXLive includes the pdfTeX tool, which can be used to convert the LaTeX output to a PDF file.

In the next section, we'll see how to write effective docstrings.

Writing effective docstrings

When writing docstrings, we need to focus on the essential information that our audience needs. When we look at using a library module, what do we need to know? Whatever questions we ask means that other programmers will often have similar questions. There are two boundaries that we should stay inside when we write docstrings:

- It's best to avoid abstract overviews, high-level requirements, user stories, or background that is not tied directly to the code. We should provide the background in a separate document. A tool such as Sphinx can combine background material and code in a single document.
- It's best to also avoid overly detailed *how it works* implementation trivia. The code is readily available, so there's no point in recapitulating the code in the documentation. If the code is too obscure, perhaps it should be rewritten to make it clearer.

Perhaps the single most important thing that developers want is a working example of how to use the Python object. The RST `::` literal block is the backbone of these examples.

We'll often write RST code samples in the following manner:

```
Here's an example::

    d = Deck()
    c = d.pop()
```

The double colon, `::`, precedes an indented block. The indented block is recognized by the RST parser as code and will be literally passed through to the final document.

In addition to an example, the formal API is also important. We'll take a look at several API definition techniques in the later section. These rely on the RST *field list* syntax. It's very simple, which makes it very flexible.

Once we're past the example and the API, there are a number of other things that

compete for the third place. What else we need to write depends on the context. There appear to be three cases:

- **Files, including packages and modules**: In these cases, we're providing an overview or introduction to a collection of modules, classes, or function definitions. We need to provide a simple roadmap or overview of the various elements in the file. In the case where the module is relatively small, we might provide the doctest and code samples at this level.
- **Classes, including method functions**: This is where we often provide code samples and `doctest` blocks that explain the class API. Because a class may be stateful and may have a relatively complex API, we may need to provide rather lengthy documentation. Individual method functions will often have detailed documentation.
- **Functions**: We may provide code samples and `doctest` blocks that explain the function. Because a function is often stateless, we may have a relatively simple API. In some cases, we may avoid more sophisticated RST markup and focus on the `help()` function's documentation.

We'll take a look at each of these broad, vague documentation contexts in some detail.

Writing file-level docstrings, including modules and packages

A package or a module's purpose is to contain a number of elements. A package contains modules as well as classes, global variables, and functions. A module contains classes, global variables, and functions. The top-level docstrings on these containers can act as roadmaps to explain the general features of the package or module. The details are delegated to the individual classes or functions.

We might have a module docstring that looks like the following code:

```
Blackjack Cards and Decks
=========================

This module contains a definition of :class:`Card`,
:class:`Deck` and :class:`Shoe` suitable for Blackjack.

The :class:`Card` class hierarchy
---------------------------------

The :class:`Card` class hierarchy includes the following class definitions.

:class:`Card` is the superclass as well as being the class for number cards.
:class:`FaceCard` defines face cards: J, Q and K.
:class:`AceCard` defines the Ace. This is special in Blackjack because it creates a soft

We create cards using the :func:`card` factory function to create the proper
:class:`Card` subclass instances from a rank and suit.

The :class:`Suit` enumeration has all of the Suit instances.

::

    >>> from ch20_ex1 import cards
    >>> ace_clubs= cards.card( 1, cards.suits[0] )
    >>> ace_clubs
    'A♣'
    >>> ace_diamonds= cards.card( 1, cards.suits[1] )
    >>> ace_clubs.rank ==  ace_diamonds.rank
    True

The :class:`Deck` and :class:`Shoe` class hierarchy
---------------------------------------------------

The basic :class:`Deck` creates a single 52-card deck.
The :class:`Shoe` subclass creates a given number of decks.
A :class:`Deck` can be shuffled before the cards can be
extracted with the :meth:`pop` method.
A :class:`Shoe` must be shuffled and
```

```
*burned*. The burn operation sequesters a random number of cards
based on a mean and standard deviation. The mean is a number of
cards (52 is the default.)
The standard deviation for the burn is
also given as a number of cards (2 is the default.)
```

Most of the text in this docstring provides a roadmap to the contents of this module. It describes the class hierarchies, making it slightly easier to locate a relevant class. References to classes use RST inline markup. There is a role prefix followed by the reference; for example, `:class:`Card`` generates text that will be a hyperlink to the definition of the `Card` class. We'll look at these references later. In a purely Python environment, these simple references work nicely; in a polyglot environment, or when using Sphinx outside Python, there are some important variations on the role names.

The docstring includes a simple example of the `card()` factory function based on `doctest`. This advertises this function as an important feature of the module as a whole. It might make sense to provide the `doctest` explanation of the `Shoe` class, as that's perhaps the most important part of this module.

This docstring includes some inline RST markup to put class names into a monospaced font. The section titles are *underlined* with `===` and `---` lines. The RST parser can determine that the heading underlined with `===` is the parent of the headings underlined with `---`.

We'll look at using Sphinx to produce the documentation in a later section. Sphinx will leverage the RST markup to produce great-looking HTML documentation.

Now, let's look at how to write API details in RST markup.

Writing API details in RST markup

One of the benefits of using the RST markup is that we can provide formal API documentation. The API parameters and return values are formatted using an RST *field list*. Generally, a field list has the following form:

```
:field1: some value
:field2: another value
```

A field list is a sequence of field labels (as `:label:`) and a value associated with that label. The label is generally short, and the value can be as long as needed. Field lists are also used to provide parameters to directives.

We'll use an extended form of the RST field list syntax to write the API documentation. We'll extend the field name to become a multipart item. We'll add prefixes with keywords such as `param` or `type`. The prefix will be followed by the parameter's name.

There are several field prefixes. We can use any of these: `param`, `parameter`, `arg`, `argument`, `key`, and `keyword`. For example, we might write the following code:

```
:param rank: Numeric rank of the card
:param suit: Suit of the card
```

We generally use `param` (or `parameter`) for the positional parameters, and `key` (or `keyword`) for the keyword parameters. These field list definitions will be collected in an indented section. The Sphinx tool will also compare the names in the documentation with the names in the function argument list, to be sure that they match.

We can also define the type of a parameter using `type` as a prefix:

```
:type rank: integer in the range 1-13.
```

While this can be helpful, it's far better to use proper type hints in the function and method definitions. The type hints are used by Sphinx and other tools. They are checked by `mypy`, also.

For functions that return a value, we should describe the result. We can

summarize the return value with the field label of `returns` or `return`.

```
:returns: soft total for this card
```

Additionally, we should also include information about exceptions that are unique to this function. We have four aliases for this field: `raises`, `raise`, `except`, and `exception`. We would write the following code:

```
:raises TypeError: rank value not in range(1, 14).
```

We can also describe the attributes of a class. For this, we can use `var`, `ivar`, or `cvar`. We might write the following code:

```
:ivar soft: soft points for this card; usually hard points, except for aces.
:ivar hard: hard points for this card; usually the rank, except for face cards.
```

We should use `ivar` for instance variables, and `cvar` for class variables. However, there's no visible difference in the final HTML output.

These field list constructs are used to prepare docstrings for classes, class methods, and standalone functions. We'll look at each case in the later section.

Now, let's see how to write class and method function docstrings.

Writing class and method function docstrings

A class will often contain a number of elements, including attributes and method functions. A stateful class may also have a relatively complex API. Objects will be created, undergo changes in state, and possibly be garbage-collected at the end of their lives. We might want to describe some (or all) of these state changes in the class docstring or the method function docstrings.

We'll use the field list technique to document the class variables in the overall class docstring. This will generally focus on using the `:ivar variable:`, `:cvar variable:`, and `:var variable:` field list items.

Each individual method function will also use field lists to define the parameters and return the values and exceptions raised by each method function. Here's how we might start to write a class with docstrings for the class and method functions:

```
class Card:
    """
    Definition of a numeric rank playing card.
    Subclasses will define :py:class:`FaceCard` and :py:class:`AceCard`.

    :ivar rank: int rank of the card
    :ivar suit: Suit suit of the card
    :ivar hard: int Hard point total for a card
    :ivar soft: int Soft total; same as hard for all cards except Aces.
    """

    def __init__(
        self, rank: int, suit: Suit, hard: int, soft: Optional[int] = None
    ) -> None:
        """Define the values for this card.

        :param rank: Numeric rank in the range 1-13.
        :param suit: Suit object from :class:`Suits`
        :param hard: Hard point total (or 10 for FaceCard or 1 for AceCard)
        :param soft: The soft total for AceCard, otherwise defaults to hard.
        """
        self.rank = rank
        self.suit = suit
        self.hard = hard
        self.soft = soft if soft is not None else hard

    def __str__(self) -> str:
        return f"{self.rank}{self.suit}"
```

```
    def __repr__(self) -> str:
        return f"{self.__class__.__name__}(rank={self.rank}, suit={self.suit})"
```

When we include this kind of RST markup in the docstring, then a tool such as Sphinx can format very nice-looking HTML output. We've provided you with both class-level documentation of the instance variables as well as method-level documentation of the parameters to one of the method functions.

This example uses the text `:py:class:`Card`` to generate a reference to the class card. The role name in this markup is a complex-looking `:py:class:` to help distinguish the Python language domain. In complex projects, there may be multiple language domains, and the role names can reflect the variety of domains.

When we look at this class with `help(Card)`, the RST markup will be visible. It's not too objectionable, as it's semantically meaningful. This points out a balance that we may need to strike between the `help()` text and the Sphinx documents.

Writing function docstrings

A function docstring can be formatted using field lists to define the parameters and return the values and raised exceptions. Here's an example of a function that includes a docstring:

```
def card(rank: int, suit: Suit) -> Card:
    """
    Create a :py:class:`Card` instance from rank and suit.

    :param suit: Suit object
    :param rank: Numeric rank in the range 1-13
    :returns: :py:class:`Card` instance
    :raises TypeError: rank out of range

    >>> from Chapter_20.ch20_ex1 import card
    >>> str(card(3, Suit.Heart))
    '3♥'
    >>> str(card(1, Suit.Heart))
    'A♥'
    """
    if rank == 1:
        return AceCard(rank, suit, 1, 11)
    elif 2 <= rank < 11:
        return Card(rank, suit, rank)
    elif 11 <= rank < 14:
        return FaceCard(rank, suit, 10)
    else:
        raise TypeError
```

The docstring for this function includes parameter definitions, return values, and the raised exceptions. There are four individual field list items that formalize the API. We've included a `doctest` sequence as well. When we document this module in Sphinx, we'll get very nice-looking HTML output. Additionally, we can use the `doctest` tool to confirm that the function matches the simple test case.

Sphinx will expand the type hints slightly. The preceding code uses the source, `card(rank: int, suit: Suit) -> Card`. This will be expanded to `ch20_ex1.card(rank: int, suit: ch20_ex1.Suit) -> ch20_ex1.Card` in the HTML page that's created by Sphinx. The class names have a prefix added to help readers understand the software.

More sophisticated markup techniques

There are some additional markup techniques that can make a document easier to read. In particular, we often want useful cross-references between class definitions. We may also want cross-references between sections and topics within a document.

In *pure* RST (that is, without Sphinx), we need to provide proper URLs that reference different sections of our documents. We have three kinds of references:

- **Implicit references to section titles**: We can use `` `Some Heading`_ `` to refer to the `Some Heading` section. This will work for all the headings that docutils recognizes.
- **Explicit references to targets**: We can use `target_` to reference the location of `_target` in the document.
- **Inter-document references**: We have to create a full URL that explicitly references a section title. Docutils will translate section titles into all lowercase, replacing the punctuation with `-`. This allows us to create a reference to a section title in an external document like this: `` `Design <file:build.py.html#design>`_ ``.

When we use Sphinx, we get more inter-document, cross-reference capabilities. These extensions to basic RST allow us to avoid trying to write detailed URLs. To do this, we'll include a target label in front of a heading, and use the ``:ref:`label` `` syntax to refer to the label.

We might have one document with some RST, as shown in the following code snippet:

```
.. _user_stories:
User Stories
============

The user generally has three tasks: customize the simulation's parameters, run a simulat
```

Note that the label begins with `_` to show it's a target; the label itself follows the

leading _. The `..` at the beginning of the line, and the `:` make this a directive to RST. The rules of the RST process specifically stipulate ignoring unknown directives. RST can be processed by many tools, and tool-specific directives are common. One tool will quietly ignore directives intended for a different tool. For that reason, this label-as-directive technique works very elegantly.

In a separate document, we might have ``:ref:`user_stories` ``. Note the lack of a leading _; this is a reference to the label, not a definition of a label. Sphinx tracks all of the labels and the associated titles so that it can create a proper HTML reference.

Now, let's see how to use Sphinx to produce documentation.

Using Sphinx to produce the documentation

The Sphinx tool produces very good-looking documentation in a variety of formats. It can easily combine documentation from source code as well as external files with additional design notes, requirements, or background.

The Sphinx tool can be found at `http://sphinx-doc.org`. The download can become complex because Sphinx depends on several other projects. The Sphinx tutorial is outstanding.

Most projects will start by using `sphinx-quickstart` to create the initial set of files. Once the files are available, details can be added. To finalize the documentation, the `sphinx-build` program will be used.

Often, running `sphinx-build` is handled via the **make** program, which slightly simplifies the command-line use of Sphinx. In some cases, **make** may not be available.

- macOS users won't have **make** available by default. It's part of the XCode package of developer tools from Apple. While the download is very large, the XCode tools are easy to install and use. Instead of XCode, some people like to use Homebrew to install **make**. Visit `https://brew.sh` for information on the Homebrew tool. The `brew install make` command will create a useful make utility for macOS.
- For Window users, `sphinx-quickstart` will create a `make.bat` script that behaves like the **make** utility.

The **make** utility is not required, but it's convenient. We can always use `sphinx-build` directly.

Let's learn how to use Sphinx quickstart in the next section.

Using Sphinx quickstart

The handy feature of `sphinx-quickstart` is that it populates the rather complex `config.py` file via an interactive question-and-answer session. Here's a part of one such session that shows how the dialog looks; we've highlighted a few responses where the defaults don't seem to be optimal.

For more complex projects, it's simpler in the long run to separate the documentation from the working code. It's often a good idea to create a `doc` directory within the overall project tree:

```
Enter the root path for documentation. > Root path for the documentation [.]: docs
```

For very small documents, it's fine to interleave the source and HTML. For larger documents, particularly documents where there may be a need to produce LaTeX and PDF, it's handy to keep these files separate from the HTML version of the documentation:

```
You have two options for placing the build directory for Sphinx output.
Either, you use a directory "_build" within the root path, or you separate
"source" and "build" directories within the root path.
> Separate source and build directories (y/N) [n]: n
```

The next batch of questions identifies specific add-ons. It starts with the following note:

```
Please indicate if you want to use one of the following Sphinx extensions:
```

We'll suggest a set of add-ons that seem most useful for general Python development. For first-time users of Sphinx, this will be enough to get started and produce excellent documentation. Clearly, specific project needs and objectives will override these generic suggestions.

We'll almost always want to include the `autodoc` feature to produce the documentation from the docstrings. If we're using Sphinx to produce the documentation outside of the Python programming, we may want to turn `autodoc` off:

```
> autodoc: automatically insert docstrings from modules (y/N) [n]: y
```

If we have `doctest` examples, we can have Sphinx run the doctest for us. For small projects, where most of the testing is done via `doctest`, this can be very handy. For larger projects, we'll often have a unit test script that includes doctest. Performing the doctest via Sphinx, as well as through the formal unit test, is still a good idea:

```
> doctest: automatically test code snippets in doctest blocks (y/N) [n]: y
```

A mature development effort may have many projects that are closely related; closely-related projects might have multiple, related Sphinx documentation directories:

```
> intersphinx: link between Sphinx documentation of different projects (y/N) [n]:
```

The `todo` extension allows us to include a `.. todo::` directive in our docstrings. We can then add a special `.. todolist::` directive to create an official to-do list in the documentation:

```
> todo: write "todo" entries that can be shown or hidden on build (y/N) [n]:
```

The coverage report could be a handy quality assurance metric:

```
> coverage: checks for documentation coverage (y/N) [n]:
```

For projects that involve any math, having a LaTeX toolset allows us to have the math nicely typeset as graphic images and included in HTML. It also leaves the raw math in the LaTeX output. MathJax is a web-based JavaScript library that also works in the following manner:

```
> pngmath: include math, rendered as PNG images (y/N) [n]: y
> mathjax: include math, rendered in the browser by MathJax (y/N) [n]:
```

For very complex projects, we might need to produce the variant documentation:

```
> ifconfig: conditional inclusion of content based on config values (y/N) [n]:
```

Most application documentations describe an API. We should include both the `autodoc` and `viewcode` features. The `viewcode` option allows the reader to view the source so that they can understand the implementation in detail:

```
> viewcode: include links to the source code of documented Python objects (y/N) [n]: y
```

Finally, when working with GitHub, it helps to include a special file to prevent

using the Jekyll tools to render the pages.

```
> githubpages: create .nojekyll file to publish the document on GitHub pages (y/n) [n]:
```

The `autodoc` and `doctest` features mean that we can focus on writing docstrings within our code. We only need to write very small Sphinx documentation files to extract the docstring information. For some developers, the ability to focus on the code reduces the fear factor associated with writing the documentation.

In the next section, we'll see how to write the Sphinx documentation.

Writing Sphinx documentation

It's often helpful to start with an outline of placeholders for the documentation that will accumulate as the software development proceeds. One structure that might be helpful is based on the 4+1 views of an architecture. For more information, refer to *The Software Architects Handbook* available from Packt Publishing.

We can create five top-level documents under our `index.html` root: `user_stories`, `logical`, `process`, `implementation`, and `physical`. Each of these needs an RST title paragraph, but nothing more is required.

We can then update the `.. toctree::` directive that's generated in the Sphinx `index.rst` file by default:

```
.. Mastering OO Python documentation master file, created by
   sphinx-quickstart on Fri Jan 31 09:21:55 2014.
   You can adapt this file completely to your liking, but it should at least
   contain the root `toctree` directive.

Welcome to Mastering OO Python's documentation!
===============================================

Contents:

.. toctree::
   :maxdepth: 2

   user_stories
   logical
   process
   implementation
   physical

Indices and tables
==================

* :ref:`genindex`
* :ref:`modindex`
* :ref:`search`
```

Once we have a top-level structure, we can use the `make` command to build our documentation:

```
make doctest html
```

This will run our doctests, and, if all the tests pass, it will create the HTML

documentation.

In the next section, we'll look at filling in the details for the various views of the application software.

Filling in the 4+1 views for documentation

As the development proceeds, the 4+1 views can be used to organize the details that accumulate. The idea is to collect the information that belongs outside the narrow focus of docstrings in the code.

The `user_stories.rst` document is where we can collect user stories, requirements, and other high-level background notes. This might evolve into a directory tree if the user stories become complex.

The `logical.rst` document is a place to collect our initial OO designs for the class, module, and package. This should be the origin of our design thinking. It might contain alternatives, notes, mathematical backgrounds, proofs of correctness, and diagrams of the logical software design. For relatively simple projects, where the design is relatively clear, this may remain empty. For complex projects, this may describe some sophisticated analysis and design that serves as the background or justification for the implementation.

The final OO design will be the Python modules and classes that belong in the `implementation.rst` file. We'll take a look at this in a little more detail, as this will become our API documentation. This part will be based in a direct way on our Python code and the RST-markup docstrings.

The `process.rst` document can collect information about the dynamic, runtime behavior. This would include topics such as concurrency, distribution, and integration. It may also contain information on performance and scalability. The network design and protocols used might be described here.

For smaller applications, it may not be perfectly clear what material should go into the process document. This document may overlap with the logical design and the overall architectural information. When in doubt, we have to strive for clarity based on the audience's need for information. For some users, many small documents are helpful. For other users, a single large document is preferred.

The `physical.rst` file is where the deployment details can be recorded. A description of the configuration details would go here, specifically, the environment variables, the configuration file format details, the available logger names, and other information required for administration and support. This may also include configuration information such as server names, IP addresses, account names, directory paths, and related notes. In some organizations, an administrator may feel that some of these details are not appropriate for general software documentation.

In the next section, we'll see how to write the implementation document.

Writing the implementation document

The `implementation.rst` document can be based on the use of `automodule` to create the documentation. Here's how an `implementation.rst` document might start:

```
Implementation
==============

Here's a reference to the `inception document <_static/inception_doc/index.html>`_

Here's a reference to the :ref:`user_story`

The ch20_ex1 module
-------------------

..  automodule:: ch20_ex1
    :members:
    :undoc-members:
    :special-members:

Some Other Module
-----------------

We'd have an ``..  automodule::`` directive here, too.
```

We used two kinds of RST headings: there's a single, top-level heading, and two subheadings. RST deduces the relationship between the parent and the children. In this example, we've used `===` for the parent heading (also the title) and `---` for the subheadings.

We've provided you with an explicit reference to a document that was copied into the `_static` directory as `inception_doc`. We created a sophisticated RST link from the words *inception document* to the actual document's `index.html` file.

`:ref:`user_story`` is an internal cross-reference to another section. Sphinx tracks all of the titles and headings that are preceded by directives used as labels. In this case, there will be title with `.. _user_story:` on the line in front of the title. `_user_story` is an RST target; it must begin with _. The title following the target will be referred to by the `:ref:`user_story`` link. We can change the title, and the HTML will be updated properly.

Within the two subheadings, we used the Sphinx `.. automodule::` directive to

extract the docstrings from two modules. We've provided you with three parameters to the automodule directives:

- `:members:`: This includes all the members of the module. We can list explicit member classes and functions, instead of listing all the members.
- `:undoc-members:`: This includes members who lack proper docstrings. This is handy when starting development; we'll still get some API information, but it will be minimal.
- `:special-members:`: This includes special-method name members, which are not included in the Sphinx documentation by default.

This gives us a relatively complete view. If we leave out all of these parameters, `:undoc-members:` and `:special-members:`, we'll get a smaller, more focused document.

Our `implementation.rst` can evolve as our project evolves. We'll add the `automodule` references as the modules are completed.

The organization of the `.. automodule::` directives can provide us with a useful roadmap or overview of a complex collection of modules or packages. A little time spent organizing the presentation so that it shows us how the software components work together is more valuable than a great deal of verbiage. The point is not to create great narrative literature; the point is to provide guidance to the other developers.

Creating Sphinx cross-references

Sphinx expands the cross-reference techniques available via RST. The most important set of cross-reference capabilities is the ability to directly refer to specific Python code features. These make use of the inline RST markup using the `:role:`text`` syntax. In this case, a large number of additional roles are part of Sphinx.

We have the following kinds of cross-reference roles available:

- The `:py:mod:`some_module`` syntax will generate a link to the definition of this module or package.
- The `:py:func:`some_function`` syntax will generate a link to the definition of the function. A qualified name with `module.function` or `package.module.function` can be used.
- The `:py:data:`variable`` and `:py:const:`variable`` syntax will generate a link to a module variable that's defined with a `.. py:data:: variable` directive. A *constant* is simply a variable that should not be changed.
- The `:py:class:`some_class`` syntax will link to the class definition. Qualified names such as `module.class` can be used.
- The `:py:meth:`class.method`` syntax will link to a method definition.
- The `:py:attr:`class.attribute`` syntax will link to an attribute that's defined with a `.. py:attribute:: name` directive.
- The `:py:exc:`exception`` syntax will link to a defined exception.
- The `:py:obj:`some_object`` syntax can create a generic link to an object.

If we use ` ``SomeClass`` ` in our docstring, we'll get the class name in a monospaced font. If we use `:py:class:`SomeClass``, we get a proper link to the class definition, which is often far more helpful.

The `:py:` prefix on each role is there because Sphinx can be used to write the documentation about other languages in addition to Python. By using this `:py:` prefix on each role, Sphinx can provide proper syntax additions and highlighting.

Here's a docstring that includes explicit cross-references to other classes and

exceptions:

```
def card(rank: int, suit: Suit) -> Card:
    """
    Create a :py:class:`Card` instance from rank and suit.
    Can raise :py:exc:`TypeError` for ranks out of the range 1 to 13, inclusive.

    :param suit: Suit object
    :param rank: Numeric rank in the range 1-13
    :returns: :py:class:`Card` instance
    :raises TypeError: rank out of range
    """
```

By using `:py:class:`Card`` instead of ``` ``Card`` ```, we're able to create explicit links between this comment block and the definition of the `Card` class. While it's a bit of typing overhead to include the role prefix, the resulting web of cross-references leads to very useful documentation.

When our projects get larger, we may need to use directories to organize the files. In the next section, we'll look at techniques for refactoring a long flat sequence of files into directories to impose some structure and organization.

Refactoring Sphinx files into directories

For larger projects, we'll need to use directories instead of simple files. It's common to have a project expand from a few simple files to a more complex structure, where the files are replaced by directories. In this case, we'll perform the following steps to refactor an RST file into a directory that contains RST files:

1. Add the directory – `implementation`, for example.
2. Move the original `implementation.rst` to `implementation/index.rst`.
3. Change the original `index.rst` file. Switch the `.. toctree::` directive to reference `implementation/index` instead of `implementation`.

We can then work within the `implementation` directory using the `.. toctree::` directive in the `implementation/index.rst` file to include other files that are in this directory.

When our documentation is split into simple directories of simple text files, we can edit small, focused files. Individual developers can make significant contributions without encountering any file-sharing conflicts that arise when trying to edit a word-processing document.

Handling legacy documents

A software development project often starts with a collection of documents describing the reasons for the project. These documents can be called *inception* documents because they describe the project at its inception. These documents may include memos and presentations justifying the project. They often describe the most essential user stories.

Ideally, these inception documents are already text files. Pragmatically, they're almost never text. Often, the inception document is a slideshow format, perhaps Microsoft PowerPoint, Apple Keynote, or Google Slides. These documents mix text with diagrams, making them a challenge to work with.

The goal is to create a text file that can be part of the RST-based documentation. For example, many word processors can save a document as text. Some word processors can produce Markdown markup; this is easily converted to RST. In some cases, we might want to use a tool such as pandoc to extract RST markup from the original document. This lets us import the document into the project and work with it via the Sphinx tools. For more information on pandoc, refer to https://pandoc.org.

One of the difficult cases is a project where some of the inception documentation is a slideshow. The diagrams and images are first-class parts of the content and don't have handy text representations. In these cases, it's sometimes best to export the presentation as an HTML document and put this into the Sphinx doc/source/_static directory. This will allow us to integrate the original material into Sphinx via simple RST links of the `Inception <_static/inception_doc/index.html>`_ form.

In some cases, an interactive, web-based tool, such as Trello, is used to manage the project or user stories. In this case, the inception and background documentation can be handled via simple URL references of this form: `Background <http://someservice/path/to/page.html>`_.

In the next section, we'll see how to write the documentation.

Writing the documentation

An important part of software quality comes from noting that the final product of software development is far more than the code. As we noted in `Chapter 17`, *Designing for Testability*, code that cannot be trusted cannot be used. In that chapter, we suggested testing was essential to establishing trust. We'd like to generalize that a bit. In addition to detailed testing, there are several other quality attributes that make the code usable. Trustworthiness supports usability.

There are a number of aspects of trustworthy code:

- We understand the use cases
- We understand the data model and processing model
- We understand the test cases

When we look at more technical quality attributes, we see that these are really about understanding. For example, debugging seems to mean that we can confirm our understanding of how the application works. Auditability also seems to mean that we can confirm our understanding of processing by viewing specific examples to show that they work as expected.

Documentation creates trust. For more information on the software quality, start here: `https://s-cube-network.eu/km/qrm/index.html`. There is a lot to learn about software quality; it's a very large area, and this is only one small aspect.

One way to create detailed documentation is to produce both the final human-readable document and the working source code from the same source. This is the idea of literate programming, which is the subject of the next section.

Literate programming

The idea of separating *documentation* from *code* can be viewed as an artificial distinction. Historically, we wrote documentation outside the code because the programming languages were relatively opaque and biased toward efficient compilation rather than clear exposition. Different techniques have been tried to reduce the distance between the working code and documentation pertaining to the code. Embedding more sophisticated comments, for example, is a long-standing tradition for reducing the distance between code and notes. Python takes this a step further by including a formal docstring in packages, modules, classes, and functions.

The literate programming approach to software development was pioneered by Donald Knuth. The idea is that a single source document can produce efficient code, as well as good-looking documentation. For machine-oriented assembler languages, and languages such as C, there's an additional benefit of moving away from the source language – a notation that emphasizes translation – toward a document that emphasizes clear exposition. Additionally, some literate programming languages act as a higher-level programming language; this might be appropriate for C or Pascal, but it is decidedly unhelpful for Python.

Literate programming is about promoting a deeper understanding of the code. In the case of Python, the source starts out very readable. Sophisticated literate programming isn't required to make a Python program understandable. Indeed, the main benefit of literate programming for Python is the idea of carrying deeper design and use case information in a form that is more readable than simple Unicode text.

For more information, refer to `http://www.literateprogramming.com` and `http://xml.coverpages.org/xmlLitProg.html`. The book *Literate Programming*, by Donald Knuth, is *the* seminal title on this topic.

Use cases for literate programming

There are two essential goals when creating a literate program:

- **A working program**: This is the code, extracted from the source document(s) and prepared for the compiler or interpreter.
- **Easy-to-read documentation**: This is the explanation, plus the code, plus any helpful markup prepared for the presentation. This document could be in HTML, ready to be viewed, or it could be in RST, and we'd use docutils `rst2html.py` to convert it to HTML. Alternatively, it could be in LaTeX and we run it through a LaTeX processor to create a PDF document.

The *working program* goal means that our literate programming document will cover the entire suite of the source code files. While this seems daunting, we have to remember that well-organized code snippets don't require a lot of complex hand-waving; in Python, code itself can be clear and meaningful.

The *easy-to-read documentation* goal means that we want to produce a document that uses something other than a single font. While most code is written in a monospaced font, it isn't the easiest on our eyes. The essential Unicode character set doesn't include helpful font variants such as bold or italic either. These additional display details (the font change, size change, style change) have evolved over the centuries to make a document more readable.

In many cases, our Python IDE will color-code the Python source. This is helpful, too. The history of written communication includes a lot of features that can enhance readability, none of which are available in simple Python source using a single font.

Additionally, a document should be organized around the problem and the solution. In many languages, the code itself *cannot* follow a clear organization because it's constrained by purely technical considerations of syntax and the order of the compilation.

Our two goals boil down to two technical use cases:

- Converting an original source text into the code.
- Converting an original source text into the final documentation.

We can, to an extent, refactor these two use cases in some profound ways. For example, we can extract the documentation from the code. This is what the `pydoc` module does, but it doesn't handle the markup very well.

Both versions, code and final document, can be made isomorphic. This is the approach taken by the PyLit project. The final documentation can be embedded entirely in Python code via docstrings as well as `#` comments. The code can be embedded entirely in RST documents using `::` literal blocks.

The next section shows how to work with a literate programming tool.

Working with a literate programming tool

Many **Literate Programming** (**LP**) tools are available. The essential ingredient, which varies from tool to tool, is the high-level markup language that separates the explanation from the code.

The source files that we write will contain the following three things:

- Text with markup that constitutes the explanation and the description
- Working Code in Python
- High-level markup to separate the text (with markup) from the code

Because of the flexibility of XML, this can be used as the high-level markup for literate programming. It's not easy to write, however. There are tools that work with a LaTeX-like markup based on the original web (and later CWeb) tools. There are some tools that work with RST as the high-level markup.

The essential step in choosing a tool, then, is to take a look at the high-level markup that is used. If we find that the markup is easy to write, we can comfortably use it to produce the source document.

Python presents an interesting challenge. Because we have RST-based tools such as Sphinx, we can have very literate docstrings. This leads us to two tiers of documentation:

- Explanations and background. This is outside the code. It provides supporting information on the design decisions that helped to organize the code.
- Reference and API. This is inside the Python docstrings.

This leads to a pleasant, evolutionary approach to literate programming:

- Initially, we can start by embedding the RST markup in our docstrings. A Sphinx-produced document looks good and provides a tidy explanation for the implementation choices.

- We can step beyond the docstrings to create the background documentation. This might include information on the design decisions, architecture, requirements, and user stories. In particular, descriptions of non-functional quality requirements belong outside the code.
- Once we've started to formalize this higher-level design documentation, we can more easily pick an LP tool. This tool will then dictate how we combine the documentation and code into a single, overall documentation structure. We can use an LP tool to extract the code and produce the documentation. Some LP tools can be used to run the test suite too.

Our goal is to create software that is not only well designed, but also trustworthy. As noted previously, we create trust in a number of ways, including providing a tidy, clear explanation of why our design is good.

We'll show some examples using PyLit3. For more information, see `https://pypi.org/project/pylit3/3.1.1/`. The conversion from RST to HTML can be done with Python's docutils package, specifically, the **rst2html.py** script. The math typesetting can be handled through MathJax or one of the TeX tools. Refer to `https://www.mathjax.org` or `https://www.tug.org/texlive/`.

If we use a tool such as PyLit3, we might create RST files that look like the following code:

```
#############
Combinations
#############

..  contents::

Definition
==========

For some deeper statistical calculations,
we need the number of combinations of *n* things
taken *k* at a time, :math:`\binom{n}{k}`.

..  math::

    \binom{n}{k} = \dfrac{n!}{k!(n-k)!}

The function will use an internal ``fact()`` function because
we don't need factorial anywhere else in the application.

We'll rely on a simplistic factorial function without memoization.

Test Case
=========
```

```
Here are two simple unit tests for this function provided
as doctest examples.

>>> from combo import combinations
>>> combinations(4,2)
6
>>> combinations(8,4)
70

Implementation
===============

Here's the essential function definition, with docstring:
::

  def combinations(n: int, k: int) -> int:
      """Compute :math:`\binom{n}{k}`, the number of
      combinations of *n* things taken *k* at a time.

      :param n: integer size of population
      :param k: groups within the population
      :returns: :math:`\binom{n}{k}`
      """

An important consideration here is that someone hasn't confused
the two argument values.
::

      assert k <= n

Here's the embedded factorial function. It's recursive. The Python
stack limit is a limitation on the size of numbers we can use.
::

      def fact(a: int) -> int:
          if a == 0: return 1
          return a*fact(a-1)

Here's the final calculation. Note that we're using integer division.
Otherwise, we'd get an unexpected conversion to float.
::

      return fact(n)//(fact(k)*fact(n-k))
```

This is a file written entirely in an RST markup. It contains some explanatory text, some formal math, and even some test cases. These provide us with additional details to support the relevant code sections. Because of the way PyLit works, we named the file `combo.py.txt`. There are three things we can do with this file:

- We can use PyLit3 to extract the code from this text file in the following manner:

```
python3 -m pylit combo.py.txt
```

This creates `combo.py` from `combo.py.txt`. This is a Python module that is ready to be

used.

- We can also use docutils to format this RST into an HTML page that provides the documentation and code in a form that we can read more easily than the original single-font text:

```
rst2html.py combo.py.txt combo.py.html
```

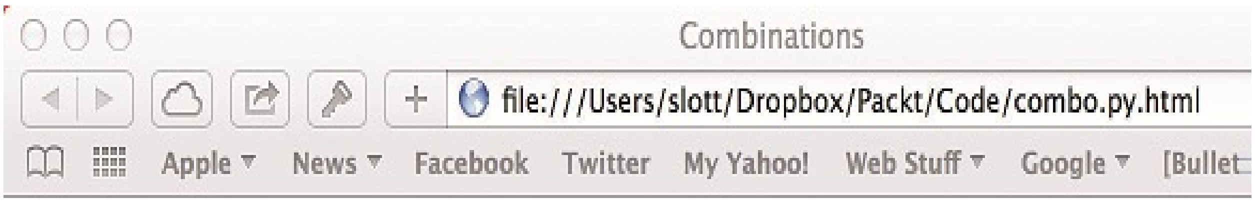
This creates `combo.py.html` ready for browsing. The `mathjax` package will be used by docutils to typeset the mathematical portions, leading to very nice-looking output.

- We can, additionally, use PyLit to run `doctest` and confirm that this program really works:

```
python3 -m pylit --doctest combo.py.txt
```

This will extract the `doctest` blocks from the code and run them through the `doctest` tool. We'll see that the three tests (the import and the two function evaluations) all produce the expected results.

The final web page produced by this would look something like the following screenshot:



Combinations

Contents

Definition

For some deeper statistical calculations, we need the number of combinations of n things taken k at a time, $\binom{n}{k}$.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

The function will use an internal `fact()` function because we don't need factorial anywhere else in the application.

We'll rely on a simplistic factorial function without memoization.

Test Case

The HTML details are handled seamlessly by the RST to HTML tool, letting us focus on simple markup and correct software.

Our goal is to create software that is trustworthy. A tidy, clear explanation of why our design is good is an important part of this trust. By writing the software and the documentation side-by-side in a single source text, we can be sure that our documentation is complete and provides a sensible review of the design decisions and the overall quality of the software. A simple tool can extract working code and documentation from a single source, making it easy for us to create the software and the documentation.

Summary

In this chapter, we looked at four ways to create usable documentation. We can incorporate the information into the docstrings in our software. We can use `pydoc` to extract the API reference information from our software. We can use Sphinx to create more sophisticated and elaborate documentation. Also, we can use a literate programming tool to create even deeper and more meaningful documentation.

Design considerations and tradeoffs

The docstring should be considered as essential as any other part of the Python source. This ensures that the `help()` function and `pydoc` will work correctly. As with unit test cases, this should be viewed as a mandatory element of the software.

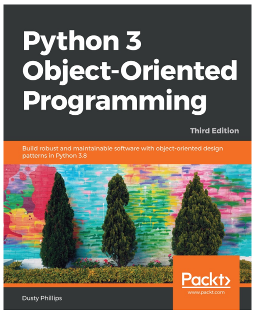
The documentation created by Sphinx can be very good looking; it will tend to parallel the Python documentation. Our objective all along has been seamless integration with the other features of Python. Using Sphinx tends to introduce an additional directory structure for the documentation source and build.

As we design our classes, the question of how to describe the design is almost as important as the resulting design itself. Software that can't be explained quickly and clearly will be viewed as untrustworthy.

Taking the time to write an explanation may identify hidden complexities or irregularities. In these cases, we might refactor a design, neither to correct a bug nor to improve performance, but to make it easier to explain. The ability to explain is a quality factor that has tremendous value.

Other Books You May Enjoy

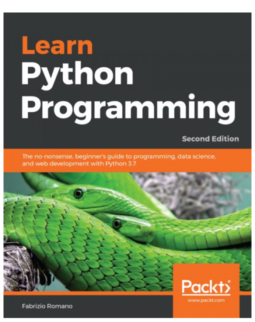
If you enjoyed this book, you may be interested in these other books by Packt:



Python 3 Object-Oriented Programming - Third Edition
Dusty Phillips

ISBN: 978-1-78961-585-2

- Implement objects in Python by creating classes and defining methods
- Grasp common concurrency techniques and pitfalls in Python 3
- Extend class functionality using inheritance
- Understand when to use object-oriented features, and more importantly when not to use them
- Discover what design patterns are and why they are different in Python
- Uncover the simplicity of unit testing and why it's so important in Python
- Explore concurrent object-oriented programming



Learn Python Programming - Second Edition
Fabrizio Romano

ISBN: 978-1-78899-666-2

- Get Python up and running on Windows, Mac, and Linux
- Explore fundamental concepts of coding using data structures and control flow
- Write elegant, reusable, and efficient code in any situation
- Understand when to use the functional or OOP approach
- Cover the basics of security and concurrent/asynchronous programming
- Create bulletproof, reliable software by writing tests
- Build a simple website in Django
- Fetch, clean, and manipulate data

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!